Amazon GameLift: Developer Guide
Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is Amazon GameLift?</td>
<td>1</td>
</tr>
<tr>
<td>Why GameLift ?</td>
<td>1</td>
</tr>
<tr>
<td>GameLift Solutions</td>
<td>1</td>
</tr>
<tr>
<td>Managed GameLift</td>
<td>1</td>
</tr>
<tr>
<td>Managed GameLift with Realtime Servers</td>
<td>2</td>
</tr>
<tr>
<td>Standalone GameLift FleetIQ [PREVIEW]</td>
<td>2</td>
</tr>
<tr>
<td>How GameLift Works</td>
<td>3</td>
</tr>
<tr>
<td>Key Components</td>
<td>3</td>
</tr>
<tr>
<td>Hosting Game Servers</td>
<td>4</td>
</tr>
<tr>
<td>Running Game Sessions</td>
<td>5</td>
</tr>
<tr>
<td>Scaling Fleet Capacity</td>
<td>6</td>
</tr>
<tr>
<td>Monitoring Fleet Activity and Troubleshooting</td>
<td>7</td>
</tr>
<tr>
<td>Networking With AWS Resources</td>
<td>8</td>
</tr>
<tr>
<td>How Realtime Servers Work</td>
<td>8</td>
</tr>
<tr>
<td>What are Realtime Servers?</td>
<td>8</td>
</tr>
<tr>
<td>Choosing Realtime Servers for Your Game</td>
<td>9</td>
</tr>
<tr>
<td>Key Components</td>
<td>9</td>
</tr>
<tr>
<td>How Realtime Servers Manages Game Sessions</td>
<td>9</td>
</tr>
<tr>
<td>How Realtime Clients and Servers Interact</td>
<td>9</td>
</tr>
<tr>
<td>Customizing a Realtime Server</td>
<td>10</td>
</tr>
<tr>
<td>Deploying and Updating Realtime Servers</td>
<td>11</td>
</tr>
<tr>
<td>How Players Connect to Games</td>
<td>12</td>
</tr>
<tr>
<td>Game and Player Session Features</td>
<td>12</td>
</tr>
<tr>
<td>How GameLift FlexMatch Works</td>
<td>13</td>
</tr>
<tr>
<td>FlexMatch Key Features</td>
<td>13</td>
</tr>
<tr>
<td>FlexMatch Components</td>
<td>14</td>
</tr>
<tr>
<td>Matchmaking Process</td>
<td>14</td>
</tr>
<tr>
<td>Backfill Process</td>
<td>15</td>
</tr>
<tr>
<td>Game Architecture with Managed GameLift</td>
<td>16</td>
</tr>
<tr>
<td>Game Architecture with Realtime Servers</td>
<td>18</td>
</tr>
<tr>
<td>Setting Up</td>
<td>21</td>
</tr>
<tr>
<td>Set Up an AWS Account</td>
<td>21</td>
</tr>
<tr>
<td>IAM Policy Examples</td>
<td>21</td>
</tr>
<tr>
<td>Set Up a Role for Amazon GameLift Access</td>
<td>22</td>
</tr>
<tr>
<td>GameLift SDKs</td>
<td>23</td>
</tr>
<tr>
<td>For Custom Game Servers</td>
<td>23</td>
</tr>
<tr>
<td>For Realtime Servers</td>
<td>24</td>
</tr>
<tr>
<td>For Client Services</td>
<td>24</td>
</tr>
<tr>
<td>SDK Compatibility</td>
<td>25</td>
</tr>
<tr>
<td>Tools and Resources</td>
<td>26</td>
</tr>
<tr>
<td>Core Tools</td>
<td>26</td>
</tr>
<tr>
<td>Additional Resources</td>
<td>26</td>
</tr>
<tr>
<td>Billing Alerts</td>
<td>27</td>
</tr>
<tr>
<td>Using GameLift in AWS Regions</td>
<td>27</td>
</tr>
<tr>
<td>Getting Started</td>
<td>29</td>
</tr>
<tr>
<td>Explore GameLift</td>
<td>29</td>
</tr>
<tr>
<td>Realtime Servers Sample Game (Full Source)</td>
<td>29</td>
</tr>
<tr>
<td>Custom Game Server Sample (Console Experience)</td>
<td>30</td>
</tr>
<tr>
<td>Get Started with Custom Servers</td>
<td>30</td>
</tr>
<tr>
<td>Get Started with Realtime Servers</td>
<td>31</td>
</tr>
<tr>
<td>Preparing Games for GameLift</td>
<td>34</td>
</tr>
<tr>
<td>Integrating Games with Custom Game Servers</td>
<td>34</td>
</tr>
<tr>
<td>Game Engines and GameLift</td>
<td>34</td>
</tr>
<tr>
<td>Integrating a Game Server</td>
<td>42</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Ports</td>
<td>175</td>
</tr>
<tr>
<td>Build</td>
<td>175</td>
</tr>
<tr>
<td>ARNs</td>
<td>175</td>
</tr>
<tr>
<td>View Game and Player Info</td>
<td>176</td>
</tr>
<tr>
<td>Game sessions</td>
<td>176</td>
</tr>
<tr>
<td>Player sessions</td>
<td>176</td>
</tr>
<tr>
<td>Player information</td>
<td>177</td>
</tr>
<tr>
<td>View Your Aliases</td>
<td>177</td>
</tr>
<tr>
<td>Alias Catalog</td>
<td>177</td>
</tr>
<tr>
<td>Alias Detail</td>
<td>177</td>
</tr>
<tr>
<td>Monitoring GameLift</td>
<td>178</td>
</tr>
<tr>
<td>Monitor with CloudWatch</td>
<td>178</td>
</tr>
<tr>
<td>Amazon GameLift Metrics for Fleets</td>
<td>178</td>
</tr>
<tr>
<td>Amazon GameLift Metrics for Queues</td>
<td>183</td>
</tr>
<tr>
<td>Amazon GameLift Metrics for Matchmaking</td>
<td>185</td>
</tr>
<tr>
<td>Dimensions for Amazon GameLift Metrics</td>
<td>187</td>
</tr>
<tr>
<td>Logging API Calls</td>
<td>188</td>
</tr>
<tr>
<td>Amazon GameLift Information in CloudTrail</td>
<td>188</td>
</tr>
<tr>
<td>Understanding Amazon GameLift Log File Entries</td>
<td>189</td>
</tr>
<tr>
<td>GameLift FleetIQ Guide [PREVIEW]</td>
<td>191</td>
</tr>
<tr>
<td>GameLift FleetIQ Features</td>
<td>191</td>
</tr>
<tr>
<td>GameLift FleetIQ logic</td>
<td>192</td>
</tr>
<tr>
<td>Key resources and components</td>
<td>195</td>
</tr>
<tr>
<td>Game Architecture</td>
<td>196</td>
</tr>
<tr>
<td>Supplementing On-Premises Hosting</td>
<td>196</td>
</tr>
<tr>
<td>Lifecycles and States</td>
<td>197</td>
</tr>
<tr>
<td>Spot Rebalancing Process</td>
<td>197</td>
</tr>
<tr>
<td>Life of a Game Server Group</td>
<td>199</td>
</tr>
<tr>
<td>Life of a Game Server</td>
<td>199</td>
</tr>
<tr>
<td>Setting Up</td>
<td>201</td>
</tr>
<tr>
<td>SDKs and Platforms</td>
<td>201</td>
</tr>
<tr>
<td>Setting up your AWS account</td>
<td>201</td>
</tr>
<tr>
<td>Integrate a Game Server</td>
<td>205</td>
</tr>
<tr>
<td>Integrate a Game Client</td>
<td>207</td>
</tr>
<tr>
<td>Create &amp; Update Game Server Groups</td>
<td>208</td>
</tr>
<tr>
<td>Create &amp; Update Game Server Groups</td>
<td>209</td>
</tr>
<tr>
<td>Best Practices</td>
<td>209</td>
</tr>
<tr>
<td>Monitor with CloudWatch</td>
<td>211</td>
</tr>
<tr>
<td>Amazon GameLift Reference Guides</td>
<td>213</td>
</tr>
<tr>
<td>Service API Reference (AWS SDK)</td>
<td>213</td>
</tr>
<tr>
<td>APIs for Managed GameLift</td>
<td>213</td>
</tr>
<tr>
<td>APIs for GameLift FleetIQ</td>
<td>216</td>
</tr>
<tr>
<td>Available Programming Languages</td>
<td>216</td>
</tr>
<tr>
<td>Realtime Servers Reference</td>
<td>217</td>
</tr>
<tr>
<td>Realtime Client API (C#) Reference</td>
<td>217</td>
</tr>
<tr>
<td>Realtime Servers Script Reference</td>
<td>227</td>
</tr>
<tr>
<td>Server SDK Reference</td>
<td>232</td>
</tr>
<tr>
<td>Server API (C++) Reference</td>
<td>232</td>
</tr>
<tr>
<td>Server API (C#) Reference</td>
<td>248</td>
</tr>
<tr>
<td>Server API (Unreal Engine) Reference</td>
<td>261</td>
</tr>
<tr>
<td>FlexMatch Reference</td>
<td>269</td>
</tr>
<tr>
<td>Rule Set Schema</td>
<td>270</td>
</tr>
<tr>
<td>Rules Language</td>
<td>276</td>
</tr>
<tr>
<td>Matchmaking Events</td>
<td>280</td>
</tr>
<tr>
<td>Document History</td>
<td>289</td>
</tr>
<tr>
<td>AWS Glossary</td>
<td>308</td>
</tr>
</tbody>
</table>
What Is Amazon GameLift?

Amazon GameLift enables developers to deploy, operate, and scale dedicated, low-cost servers in the cloud for session-based, multiplayer games. Built on AWS global computing infrastructure, GameLift helps deliver high-performance, high-reliability, low-cost game servers while dynamically scaling your resource usage to meet worldwide player demand.

Why GameLift?

Here are some of the benefits of using Amazon GameLift:

- Bring your own fully custom multiplayer game servers or use ready-to-go Realtime Servers that require minimal configuration and little or no backend experience.
- Provide low-latency player experience to support fast-action game play.
- Enhance your matchmaking services with intelligent queuing, game session placement, and match backfill.
- Reduce engineering and operational effort to deploy and operate game servers.
- Get started fast and pay as you go, with no upfront costs and no long-term commitments.
- Reduce costs by up to 90% with Spot Instances.
- Rely on Amazon Web Services (AWS), including Amazon Elastic Compute Cloud (Amazon EC2) for web-scale cloud computing resources and auto-scaling to manage your hosting capacity.

Tip

Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using sample games (p. 29).

GameLift Solutions

GameLift offers a range of solutions for game developers:

- Managed GameLift hosting for custom-built game servers.
- Managed GameLift hosting with Realtime Servers
- GameLift FleetIQ standalone feature for game hosting directly on Amazon EC2 [PREVIEW]

Managed GameLift

Amazon GameLift offers a fully managed service for deploying, operating, and scaling session-based, multiplayer game servers. GameLift replaces the work required to host your own custom game servers, including buying and setting up hardware, and managing ongoing activity, security, storage, and performance tracking. Auto-scaling capabilities provide additional protection from having to pay for more resources than you need, while making sure you always have games available for new players to join with minimal waiting.

For more information on how the managed GameLift solution works, see How Amazon GameLift Works (p. 3).
Key features

- Provide high-quality game hosting to players around the world by deploying computing resources in multiple AWS Regions.
- Deploy game servers to run on Amazon Linux or Windows Server operating systems.
- Let FleetIQ optimize the use of low-cost Spot Instances. On their own, Spot Instances are not always viable for game hosting due to potential interruptions. FleetIQ prediction algorithms find the Spot Instances that are best suited to host new game sessions.
- Use automatic scaling tools to adjust your game hosting capacity to meet actual player demand. These tools allow you to keep hosting costs in line while maintaining enough capacity to get new players into games fast.
- Build a custom matchmaking service for your game using FlexMatch. Create single-team or multi-team matches for up to 200 players.
- Manage game sessions and player sessions. Configure game session characteristics, such as maximum number of players allowed, join rules, and game-specific properties.
- Choose from a range of options to help players find suitable game sessions. Use GameLift Queues to intelligently place new game sessions across multiple regions, provide players with filtered and sorted lists of available game sessions (“list and pick”), or implement a full matchmaking system with FlexMatch.
- Analyze game performance using the Amazon GameLift console to track metrics, view game session logs, and review data on individual game sessions and player sessions.
- Set up customized health tracking for server processes to detect problems fast and resolve poor-performing processes.
- Manage your game resources using AWS CloudFormation templates for GameLift.

Managed GameLift with Realtime Servers

Use Realtime Servers to stand up games that don't need custom-built game servers. This lightweight server solution provides ready-to-go game servers that can be configured to fit your game. You can deploy game servers with anything from minimal configuration settings to custom logic that is specific to your game and players.

For more information on how the managed GameLift with Realtime Servers solution works, see How Realtime Servers Work (p. 8).

Key features

- Use GameLift management features, including auto-scaling, multi-region queues, game session placement with FleetIQ, game session logging, and metrics.
- Use GameLift hosting resources, choose the type of AWS computing hardware for your fleets. Use either Spot or On-Demand Instances.
- Take advantage of a full network stack for game client/server interaction.
- Get core game server functionality with customizable server logic.
- Make live updates to Realtime configurations and server logic. Update your Realtime server configuration at any time.
- Implement FlexMatch matchmaking.

Standalone GameLift FleetIQ [PREVIEW]

GameLift FleetIQ optimizes the use of low-cost Spot Instances for cloud-based game hosting. With this feature, you can work directly with your hosting resources in Amazon EC2 and Auto Scaling while taking
How Amazon GameLift Works

This topic provides a general overview of the managed Amazon GameLift solution. It covers the core components and describes how the service makes your multiplayer game servers available to players. If you want to learn more about what managed GameLift does and how it works, start with this topic and the following related topics. To learn about other GameLift solutions, see What Is Amazon GameLift? (p. 1).

• How Players Connect to Games (p. 12)
• Game Engines and Amazon GameLift (p. 34)
• Why Use Queues? (p. 129)
• How Amazon GameLift FlexMatch Works (p. 13)

Ready to start prepping your game for hosting on GameLift? See these Getting Started with Amazon GameLift (p. 29) topics, including integration roadmaps.

Key Components

Setting up Amazon GameLift to host your game involves working with a set of key components. The relationships between these components is illustrated in Game Architecture with Managed Amazon GameLift (p. 16).

• A **game server** is your game’s server software running in the cloud. You might have a fully custom **game server build** containing the server executables, supporting assets, libraries, and dependencies. Or, if you’re using Realtime Servers, you have a **configuration script** with some optional custom game logic. You upload your game server build or script to the Amazon GameLift service, and GameLift deploys it to virtual computing resources for hosting.

• A **game session** is a instance of your game server, running on Amazon GameLift resources, that players connect to and interact with. You define the basic characteristics of a game session, such as its life span and number of players.

• The **Amazon GameLift service** manages the computing resources to host your game server and makes it possible for players to connect to games. It regulates the resources needed to meet player demand, starts new game sessions, and handles new player requests by finding and reserving player slots in active game sessions. The service also collects metrics on player usage and server health.
A game client is your game's software running on a player's device. A game client makes requests to GameLift service about available game sessions. It also connects directly to a game session using information that it receives from the GameLift service.

Game services are additional custom services that you might create to handle special tasks related to GameLift. For example, most games use a client service to handle communication with the GameLift service, instead of having game clients call the service directly.

For a detailed description of how these components interact, see Amazon GameLift and Game Client/Server Interactions (p. 51).

Hosting Game Servers

Your uploaded game servers are hosted on Amazon GameLift virtual computing resources, called instances. You set up your hosting resources by creating a fleet of instances and deploying them to run your game server (either your custom game server or your configured Realtime Servers). You can design a fleet to fit your game's needs.

Fleet architecture

Build a fleet that suits your core requirements.

- What type of resources does your game need? – GameLift supports a range of operating systems and instance types. The instance type determines the kind of computing hardware used, including processing power, memory, and networking capacity. Note that fleet costs are based on both the type and the number of instances you use. Depending on your game's requirements, you might opt to use many small instances or fewer more powerful instances. Learn more about how to Choose Computing Resources (p. 100).

- Where do you want to run your game servers? – You set up fleets wherever you have players waiting to join your games. Each fleet is deployed to a single AWS region, but you can create fleets in as many regions as you need. See the available regions for Amazon GameLift at AWS Regions and Endpoints.

- How critical is game server reliability? – Your fleet uses either Spot instances or On-Demand instances. Spot instances (based on EC2's Spot instances) usually cost less, but may be interrupted during a game session. However, GameLift has additional safeguards that make game session interruptions extremely rare, and fleets with Spot instances are a good choice for most games. On-demand instances, in contrast, provide consistent availability but can be more expensive. They are a good option for games where players are strongly impacted if a game session is interrupted. Learn more about Spot and On-Demand instances.

- How many players do you need to support? – A fleet can have many instances, each one capable of hosting multiple simultaneous game sessions. You can add or remove instances from your fleet as needed, and you can use auto-scaling to automatically adjust as player demand shifts. Learn more about Scaling Fleet Capacity (p. 6).

Server runtime configuration

A fleet instance can run multiple processes simultaneously, and it can run any executable in your game server build. To determine how processes should run on each instance, you create a runtime configuration. The configuration specifies: (1) which executables to run, (2) how many processes of each executable to run concurrently, and (3) any launch parameters to use when starting each executable. The number of processes that an instance can run simultaneously depends on the instance's computing power (instance type) as well as the requirements of your game server build. Learn more about running multiple processes on a fleet (p. 101). Runtime configurations can be updated throughout the life of the fleet.

A runtime configuration can also affect how new game sessions are started on an instance. Some games require a lot of resources during the start-up phase, and it can be a good idea to limit the instance
resources that are used to activate new game sessions at any one time. You can specify a maximum number of simultaneous game session activations per instance. You can also place a time limit on each game session activation to quickly detect and shut down any game sessions that are failing to activate.

**Server security**

Enable PKI resource generation for a fleet. When this feature is turned on, GameLift generates a TLS certificate for the fleet and creates a DNS entry for each instance in the fleet. With these resources, your game can authenticate the client and server connection and encrypt all game client/server communication. This feature is particularly valuable when deploying mobile multi-player games. These services are provided through the AWS Certificate Manager (ACM) and are currently available at no additional cost.

**Fleet aliases**

An alias is a designation that can be transferred from one actual fleet to another, making it a convenient way to generalize a fleet location. For example, your game client needs to specify where (which fleet) to place a new game session. Using an alias lets you switch game clients from using one fleet to another without having to change your game client. There are several GameLift features where you have the option to specify a fleet or an alias. You can also create a "terminal" alias, which lets you point to content (such as a URL) instead of connecting to a server. This capability can be useful, for example, to prompt players to upgrade their clients.

**Running Game Sessions**

Once a game server build is successfully deployed to a fleet, the fleet is ready to host game sessions. To start a new game session for one or more players, your game client sends a request (via a game service) to the GameLift service. When the request is received, GameLift uses a feature called FleetIQ to place the new game session with the "best possible" fleet.

The meaning of "best possible fleet" is defined by you, based on your game priorities, when you define a game session queue. A queue creates a group of one or more fleets and defines how to choose the best fleet in the group for a new game session. Queues can contain fleets that are located in different regions. A new game session request specifies which queue to use, and GameLift may place the new game session with any available fleet in the queue. For example, you might use a queue with fleets in each of the five North American regions. With this type of multi-region queue, GameLift is more likely to find available resources, even during heavy traffic, and start up a new game session quickly. Fleet availability simply means that somewhere in the fleet there is at least one instance with a game server process that is free to host a new game session.

When choosing the best possible placement for a new game session, FleetIQ uses one of two methods:

- **Evaluate player latency** – Requests for new game sessions can include ping time for each player in the request. If this data is provided, FleetIQ evaluates it to determine which regions will deliver the lowest possible latency for the players. GameLift uses this information to place the new game session. You can also define latency policies to ensure that players are never placed in game sessions with unacceptable latency.

- **Use prioritized fleet list** – For requests that don't contain player latency data, FleetIQ places new game sessions based on the order that the fleets are listed in the queue. You can prioritize fleets in a queue by placing them at the top of the fleet list order. This method usually results in all game sessions being placed with the first fleet listed, and the remaining fleets serve as backups when the first fleet is full.

The queue is a powerful concept that can be used to solve a range of capacity and availability issues. Queues can be used to balance resource usage across regions, reduce the amount of wait time for players during unexpected spikes in demand, and mitigate region slowdowns or outages. You can also use them to create player pools that span regions, so that players in different regions can play together. Queues
are required when using FlexMatch matchmaking or GameLift Spot fleets. Learn more about how to Design a Game Session Queue (p. 128).

Once a game session is started on a fleet instance, the GameLift service delivers connection information to the game client, in the form of an IP address or DNS name and a port. Your game client uses this information to connect to your game server. Depending on how you set up your game server, on connection, it may communicate with the GameLift service to verify the player and report player connection status.

If the fleet is created with a TLS certificate, your game client and server can use it to establish a secure connection.

Scaling Fleet Capacity

Once a fleet is active and ready to host game sessions, you can adjust your fleet capacity to meet player demand. The cost of hosting is based on the amount of capacity you use, so you'll want to find a balance between making sure all incoming players can find a game and overspending on resources that sit idle.

You scale a fleet by adjusting the number of instances in it. Your runtime configuration determines how many game sessions and players each instance can host, so by scaling instances you're increasing or decreasing the availability for game sessions and players. GameLift provides a highly effective auto-scaling tool, or you can opt to manually set fleet capacity. Learn more about how to Scaling Amazon GameLift Fleet Capacity (p. 119).

Auto-scaling

With auto-scaling enabled, GameLift tracks a fleet's hosting metrics and determines when to add or remove instances based on a set of guidelines that you define. With the right auto-scaling policies in place, GameLift can adjust capacity directly in response to changes in player demand. Learn more about improving cost efficiency with automatic scaling.

There are two methods of auto-scaling available:

- **Target-based scaling** – With this method, you specify a desired outcome and GameLift scales the fleet up or down to achieve that outcome. Target tracking uses the metric "percent available game sessions", that is, the percentage of healthy server processes that are not currently hosting a game session. Available game sessions is your buffer—they represent the number of new game sessions and new players that can join a game with a minimal wait time. With target tracking, you choose the buffer size that fits your game. For example, for a game with highly volatile demand, you may need a larger buffer size. This method is the preferred option, as it is simpler and more effective for more games. Learn more about how Target Tracking works.

- **Rule-based scaling** – This method gives you more fine-grained control of scaling actions. It is also more complex to set up and manage and is more likely to have unexpected results. Each policy specifies when to trigger a scaling event and what action to take in response. For example, a policy might state: “If idle instances falls below 20 for 10 consecutive minutes, then increase capacity by 10%.” Most fleets require multiple policies to manage fleet capacity effectively, but multiple policies can have unexpected compound effects, which adds to the complexity. Learn how to Auto-Scale with Rule-Based Policies (p. 125).

Fleet scaling in action

A fleet scaling event can be triggered in several ways, either by making a change to the desired capacity through auto-scaling or manual scaling, or when instances are shut down for health or other reasons. Essentially, all scaling events are triggered when a fleet's "desired" instance count does not match its "active" instance count. This circumstance causes Amazon GameLift to add or remove instances, as needed, in order to make the active instance count match the desired instance count.
• When desired instance count exceeds active instance count, Amazon GameLift requests additional instances and, once available, begins the process of installing the game server build to the new instances and starting up the game server processes. As soon as one server process is active on an instance, the number of active instances is increased by one. Amazon GameLift continues to add instances until the two count values are even.

• When active instance count exceeds desired instance count, Amazon GameLift begins searching for instances it can remove. Any available instance (that is, not hosting any game sessions) can be terminated, as well as any non-protected instance even when hosting active game sessions. If no instances can be removed, the scale-down event fails. In this circumstance, the disparity between desired and active instance counts will continue to trigger scale-down events until an instance is free to be removed. Amazon GameLift then starts the termination process, which includes notifying all server processes on the instance to initiate a graceful shutdown. Once the instance is terminated, the number of active instances is decreased by one. Amazon GameLift continues to remove instances until the two count values are even.

**Additional scaling features**

Additional features related to fleet capacity and scaling include:

• **Game session protection** – Prevent game sessions that are hosting active players from being terminated during a scale-down event. Game session protection can be turned on fleet-wide, or it can be turned on for individual game sessions. An instance cannot be terminated if any of its server processes are hosting protected game sessions. Game sessions are not protected from termination due to health or for spot-instance-related interruptions (see *On-Demand versus Spot Instances (p. 100)*).

• **Scaling limits** – Control overall instance usage by setting minimum and maximum limits on the number of instances in a fleet. These limits apply when auto-scaling or when manually setting capacity.

• **Enabling/disabling auto-scaling** – Switch auto-scaling on or off at the fleet level without changing or deleting your auto-scaling policies. This feature allows you to temporarily scale your fleets manually when needed.

• **Scaling metrics** – Track a fleet's history of capacity and scaling events in graph form. View capacity in conjunction with fleet utilization metrics to evaluate the effectiveness of your scaling approach. The following graph shows a fleet with target tracking set to a 15% buffer; the percentage of available game session slots (in green) automatically adjusts as fleet capacity (in blue and orange) changes.

**Monitoring Fleet Activity and Troubleshooting**

Once you have fleets up and running, Amazon GameLift collects a variety of information to help you monitor the performance of your deployed game servers. Use this information to optimize your use of resources, troubleshoot issues, and gain insight into how players are active in your games.

• **Fleet, game session, and player session details** – This data includes status, which can help identify health issues, as well as details such as game session length and player connection time.

• **Utilization metrics** – Amazon GameLift tracks fleet metrics over time:
  • For instances: network activity and CPU utilization
  • For server processes: number of active processes, new activations, and terminations
  • For games and players: number of active game sessions and player sessions

• **Server process health** – Amazon GameLift tracks the health of each server process running on a fleet, including the number of healthy processes, percent of active processes that are healthy, and number of abnormal terminations.

• **Game session logs** – You can have your game servers log session data and set Amazon GameLift to collect and store the logs once the game session ends. Logs can then be downloaded from the service.
All of this data is available through the Amazon GameLift console. The console dashboard presents an overview of activity across all you builds and fleets as well as the option to drill down to more detailed information.

**Networking With AWS Resources**

In many situations, you want your hosted game servers and applications to be able to communicate with your other AWS resources. For example, you might use a set of web services to support your game, such as for player authentication or social networking. This type of communication poses a challenge due to ownership issues. When you deploy game servers using Amazon GameLift, the fleets and instances are allocated to your account but they are owned and managed by the Amazon GameLift service. As a result, to access AWS resources that are managed by your AWS account, you need to explicitly permit access by the Amazon GameLift service.

Amazon GameLift provides a couple of options for managing this type of access. Learn more about how to Access AWS Resources From Your Fleets (p. 45).

**How Realtime Servers Work**

This topic provides an overview of the managed Amazon GameLift with Realtime Servers solution. It discusses when it is a good fit for your game, and explains how Realtime Servers supports multiplayer gaming. To learn more about other GameLift solutions, see What Is Amazon GameLift? (p. 1).

**What are Realtime Servers?**

Realtime Servers are lightweight, ready-to-go game servers that are provided by GameLift for you to use with your multiplayer games. While many games need a custom game server to handle complex physics and computations, this is overkill for many other games. Since Realtime Servers eliminate the need to develop, test, and deploy a custom game server, choosing this solution can help minimize the time and effort required to complete your game.

Key features include:

- **Full network stack for game client/server interaction.** Realtime Servers makes use of TCP and UDP channels for messaging. You can also opt to use built-in server authentication and data packet encryption by enabling GameLift-generated TLS certificates.

- **Core game server functionality.** A Realtime server starts (and stops) game sessions, manages game and match data, and accepts client connections. The game server maintains a synchronized game session state by receiving game state information from each client and relaying it to other clients in the game session.

- **Integrated with the GameLift service.** A Realtime server is set up to communicate with the GameLift service, which triggers the Realtime server to start game sessions, validate players when they connect, and collects player connection status and game health state from the game server. In contrast, this functionality must be implemented in a custom game server.

- **Customizable server logic.** You can configure your Realtime servers and customize them with server-side game logic as best fits your game. Alternatively, provide a minimal configuration and to use them as simple relay servers. Learn more about Customizing a Realtime Server (p. 11).

- **Live updates to Realtime configurations and server logic.** Update your Realtime server configuration at any time. GameLift regularly checks for updated configuration scripts, so once you upload a new version, it is quickly deployed to your fleet and used with all new game sessions.

- **FlexMatch matchmaking.** Game clients that use Realtime Servers can make use of all FlexMatch matchmaking features, including for large matches.

- **Flexible control of hosting resources.** For games that are deployed with Realtime Servers, you can use all GameLift management features, including auto-scaling, multi-region queues, game session
Choosing Realtime Servers for Your Game

Choosing Realtime Servers instead of building a custom game server primarily comes down to your game's need for server complexity. Unless your game needs complicated server-side game logic, split-second computations for gameplay physics, or other custom capabilities, Realtime Servers may be the better solution for your game. Games that use Realtime Servers to best effect include lighter weight games or games that manage a higher percentage of the computational work on the game client. Examples include messaging games, turn-based strategy games, and many types of mobile games. Realtime Servers, coupled with the use of FleetIQ, provides effective tools to minimize player latency suitable for nearly all game types.

Key Components

When working with Realtime Servers, you work with the following components. Learn more about these components and how they work together in Game Architecture with Realtime Servers (p. 18).

- A Realtime server provides client/server networking for your game. It starts game sessions when triggered by the GameLift service, requests validation for players when they connect, and reports back on the status player connections and game health. The server relays game state data between all connected players, and executes custom game logic if provided.
- A game client is your game's software running on a player's device. The game client (through a client service) makes requests to the GameLift service to find game sessions to join or to start new ones, and connects to a Realtime server to participate in a game. Once connected, a game client can send and receive data, through the Realtime server, with other players in the game.
- A Realtime script provides configuration settings and optional custom game logic for your game. The script may contain minimal configuration settings or have more complex game logic. The Realtime script is deployed along with the Realtime server when starting up new hosting resources. Scripts are written in Node.js-based JavaScript.
- The GameLift service manages the computing resources needed to host your Realtime servers and makes it possible for players to connect to games. It regulates the number of resources for player demand, handles player join requests by finding and reserving player slots in active game sessions, triggers Realtime servers to start game sessions, and validates players when they connect to a game server. The service also collects metrics on Realtime server health and player usage.
- A game session is an instance of your game, run on a Realtime server. Players connect to a game session to play the game and interact with the other players.

How Realtime Servers Manages Game Sessions

GameLift manages game sessions with Realtime Servers in the same way that it handles game sessions with fully custom game servers. Players, using a game client, send requests to create new game sessions or to find and join existing game sessions. Most methods for creating game sessions, including game
session placement and FlexMatch matchmaking, are available with Realtime Servers (match backfill is not yet available).

A Realtime server, once it is deployed on a fleet of hosting instances, maintains communication with the GameLift service. The Realtime server starts a game session when it is prompted to by the GameLift service and receives available game session and player data, including matchmaking data from the service. If your game uses player sessions to reserve game slots or to authenticate player connections, the Realtime server can send a validation request to the GameLift service when the player connects. A Realtime server also reports its health status back to the GameLift service, and notifies the service when players connect/disconnect and when a game session ends. It also responds to prompts from GameLift to force a game session termination. This interaction with the GameLift service is fully built into all Realtime Servers.

You have the option of adding custom logic for game session management by building it into the Realtime script. You might write code to access server-specific objects, add event-driven logic using callbacks, or add logic based on non-event scenarios, such as a timer or status check. For example, you might want to or access game session objects, or trigger an action when a game session starts or ends.

**How Realtime Clients and Servers Interact**

During a game session, the interaction between game clients in the game is done by messaging. Game clients use messages to exchange activity, game state, and relevant game data. Game clients send messages to the Realtime server, which then relays the messages among the game clients. Game clients communicate with the server using the Realtime Client SDK, which must be integrated into your game client. The Client SDK defines a set of synchronous API calls that allow clients to connect to games, send and receive messages, and disconnect from games. It also defines a set of asynchronous callbacks, which can be implemented on the game client to enable the client to respond to certain events.

In addition, you can customize how clients and servers interact by adding game logic to the Realtime script. With custom game logic, a Realtime might implement callbacks to trigger event-driven responses. For example, when a game client notifies the server that a certain achievement is reached, the server sends a message to other game clients to prompt an announcement.

**Communication Protocol**

Communication between a Realtime server and connected game clients uses two channels: a TCP connection for reliable delivery and a UDP channel for fast delivery. When creating messages, game clients choose which protocol to use depending on the nature of the message. Message delivery is set to UDP by default. If a UDP channel is not set up or not available, all messages are sent using TCP as a fallback.

**Message Content**

Message content consists of two elements: a required operation code (opCode) and an optional payload. A message's opCode identifies a particular player activity or game event, while the payload provides additional data, as needed, related to the operation code. Both of these elements are developer-defined; that is, you define what actions map to which opCodes, and whether a message payload is needed. You game client takes action based on the opCodes in the messages it receives.

**Player Groups**

Realtime Servers provides functionality to manage groups of players. By default, all players who are connected to a game are placed in an “all players” group. In addition, developers can set up other groups for their games, and players can be members of multiple groups simultaneously. Group members can send messages to all players in the group or share game data with the group. One possible use for groups is to set up player teams and manage team communication.

**Realtime Servers with TLS Certificates**
You can opt to create Realtime Servers fleets with TLS certificate generation turned on. GameLift generates a TLS certificate for the fleet and creates a DNS entry for each instance in the fleet. This allows your game to authenticate the client/server connection and encrypt all game client/server communication. This feature makes it possible to publish games on a range of platforms, including mobile, that require enhanced security and encrypted communication. It helps to protect your game clients (and players) from server spoofing attacks, and prevents malicious actors from hacking or monitoring data transmissions. These services are provided through the AWS Certificate Manager (ACM) and are currently available at no additional cost.

With Realtime Servers, server authentication and data packet encryption is already built into the service, and is enabled when you turn on TLS certificate generation. When the game client tries to connect with a Realtime server, the server automatically responds with the TLS certificate, which the client validates. Encryption is handled using TLS for TCP (Websockets) communication and DTLS for UDP traffic.

### Customizing a Realtime Server

In its most basic form, a Realtime server performs as a stateless relay server. The Realtime server relays packets of messages and game data between the game clients that are connected to the game, but does not evaluate messages, process data, or perform any gameplay logic. Used in this way, each game client maintains its own view of the game state and provides updates to other players via the relay server. Each game client is responsible for incorporating these updates and reconciling its own game state.

Alternatively, you can customize your servers by building out the Realtime script functionality. There are many server-side processes you may choose to implement even while taking advantage of the simplicity of the Realtime Servers feature. With game logic, for example, you might opt to build a stateful game with a server-authoritative view of the game state.

A set of server-side callbacks are defined for Realtime scripts. Implement these callbacks to add event-driven functionality to your server. For example, you might:

- Authenticate a player when a game client tries to connect to the server.
- Validate whether a player can join a group when requested.
- Evaluate when to deliver messages from a certain player or to a target player, or perform additional processing in response.
- Take action, such as notifying all players, when a player leaves a group or disconnects from the server.
- Evaluate the content of game session objects or message objects and use the data.

### Deploying and Updating Realtime Servers

Realtime Servers is powered by GameLift’s dedicated server resources. There is no difference in stability and security provided. As with all servers, latency can be minimized by using GameLift’s matchmaking and queues with Fleet IQ, which optimizes game session placement based on player locations.

When deploying Realtime Servers games with GameLift, the process is nearly identical to deploying traditional game servers on GameLift. You create fleets of computing resources and deploy them with your Realtime script, which contains configuration details and optional custom logic. Using GameLift, you choose the type of fleets to use, manage fleet capacity, and control how game server processes are started and run on your fleets. The detailed description of game hosting in How Amazon GameLift Works (p. 3) represents game hosting with Realtime Servers as well as with custom game servers.

A key advantage Realtime Servers is the ability to update your scripts at any time. You do not need to create a new fleet to deploy an updated script. When you update a script, the new version is propagated to all hosting resources within a few minutes. Once the new script is deployed, all new game sessions created after that point will use the new script version (existing game sessions will continue to use the original version).
How Players Connect to Games

A *game session* is an instance of your game running on Amazon GameLift. To play your game, players can either find and join an existing game session or create a new game session and join it. Players join by creating a *player session* for the game session. If the game session is open for players—that is, it is accepting new players and has an open player slot—Amazon GameLift reserves a slot for the player and provides connection information back to the player. The player can then connect to the game session and claim the reserved slot.

For detailed information on creating and managing games sessions and player sessions, see [Add Amazon GameLift to Your Game Client](#) (p. 47).

Game and Player Session Features

Amazon GameLift provides several features related to game and player sessions:

### Host game sessions on best available resources across multiple regions

Choose from multiple options when configuring how Amazon GameLift selects resources to host new game sessions. If you’re running multiple fleets in more than one region, you can set up *game session queues* that can place a new game session on any fleet regardless of region. This feature can significantly improve the Amazon GameLift service's ability to efficiently balance resource usage and respond to changes in player demand, decreased capacity, outage events, and other issues. As a result, using queues can decrease the manual overhead needed to monitor and balance resources. You can manage queues and track queue performance metrics in the Amazon GameLift Console.

With the queues feature, you have the option of placing game sessions based on player latency information. This feature is particularly effective when supporting a matchmaking service. Requests for a new game session can also request new player sessions for one or more players. If you include latency data for each player by region, Amazon GameLift can choose a fleet in a region that provides the best possible experience for all the players.

### Control player access to game sessions

Set a game session to allow or deny join requests from new players, regardless of the number of players currently connected. You might use this feature to enable private sessions, to limit access for troubleshooting or other problem resolution, etc.

### Add custom game and player data

You can add custom data to game session and player session objects, which contain all the settings and metadata for a session. Custom data is stored with Amazon GameLift and can be retrieved by other components as needed. The Amazon GameLift service passes game session data to a game server when starting a new game session, and passes player session data to the game server when a player connects to the game session. Custom game and player data is not used by Amazon GameLift; it can be formatted as needed for use by your game client, game server, or other game services.

Game data may be useful for a variety of reasons. For example, when matching prospective players to game sessions, your game might use game properties to inform a best-match algorithm or help players choose from a list of game sessions. Alternatively, you might use game properties to pass information that a game server needs when setting up a new game session, such as a game mode or map.

Player data has a range of uses as well. For example, a matchmaking service might use player data to select a best match or team placement. A game server might customize a player’s experience based on their guild membership.
Filter and sort available game sessions

Use session search and sort to find the best match for a prospective player or allow players to browse a list of available game sessions. With this feature, you can effectively lead players to sessions that are most likely to result in a positive gaming experience. For example, if your game requires a minimum number of players, directing new players into nearly-filled games will minimize wait time for all players. Alternatively, you'll likely want to hide sessions that are nearly finished. Session search can be very useful for implementing a "join now" feature backed by a well-formulated search and sort expression that gets players into positive gaming experiences fast. Use session search and sort to find game sessions based on characteristics like session age, available player slots, current player count, maximum players allowed, and custom game session data. You can also search and sort based on your own custom game data.

Track game and player usage data

Have Amazon GameLift automatically store logs for completed game sessions. Set up log storage when integrating Amazon GameLift into your game servers. You can access stored logs by downloading them through the Amazon GameLift console or programmatically with the AWS SDK for Amazon GameLift.

Use the Amazon GameLift console to view detailed information on game sessions, including session metadata and settings as well as player session data. For each game session, you can view a list of player sessions along with total times played. You can also view metrics data and graphs that track the number of active game sessions and player sessions over time. See more information at View Data on Game and Player Sessions (p. 176) and Metrics (p. 173).

How Amazon GameLift FlexMatch Works

This topic provides an overview of the FlexMatch matchmaking system, which is available as part of the managed GameLift solutions. This topic describes the key features, components, and how the matchmaking process works. For detailed help with adding FlexMatch to your game, including how to set up a matchmaker and customize player matching, see Adding FlexMatch Matchmaking (p. 69).

GameLift FlexMatch is a customizable matchmaking service. It offers flexible tools that let you manage the full matchmaking experience in a way that best fits your game. With FlexMatch, you can build teams for your game matches, select compatible players, and find the best available hosting resources for an optimum player experience. You can also use FlexMatch backfill to find new players for existing games, so that games stay filled with compatible players throughout the life of the game session, for the best possible player experience.

With FlexMatch you can create and run as many matchmakers to fit your game modes and your players. For example, you would likely have different matchmakers to build teams for a free-for-all and a cage match.

FlexMatch Key Features

- **Customize player matching.** Design and build the types of multiplayer experiences that your players will find most compelling. For each game mode, define the team structure and set other game attributes. Build a set of custom rules to evaluate player attributes (such as skill level or role) and form the best possible player matches for a game. Use these rules to group players for new matches or find players to fill open slots in existing matches ("match backfill").

- **Create large matches.** FlexMatch can be used to create very large matches--between 41 and 200 players--using a large-match algorithm that streamlines the matching process. When creating large matches, you can choose between placing a higher priority on creating larger matches with more similar players or creating matches where all players have the best possible player latency experience.

- **Get player acceptance.** Require all players to accept a proposed match before starting. If this feature is enabled, FlexMatch waits for all players assigned to a match to accept it before the match begins.
Support player parties. Generate matches for a group of players who want to play together on the same team. Find additional players to fill out the match as needed.

- **Match players based on latency.** Use player latency information to ensure that matched players have similar response times. This feature avoids disparities in gameplay that might give some players undue advantage. It is particularly valuable when creating matches that span multiple geographic areas.

- **Relax player matching rules over time.** Strike the right balance between creating the best possible player matches and getting players into good matches fast. You decide where and when to relax strict matching rules in order to get players into games with minimal wait time.

- **Find the best hosting resources.** Use game and player information to select the best available resources to host the match for optimal gameplay experience.

- **Keep games filled with matched players.** Use FlexMatch backfill to fill empty player slots with well-matched new players throughout the life span of the game session. You can opt to enable automatic backfill or add code to your game to manage backfill manually.

### FlexMatch Components

Amazon GameLift FlexMatch requires these three key components to work together:

- **Mechanisms to trigger player matchmaking.** One mechanism determines when to initiate matchmaking for players. A second (optional) mechanism determines when to find new players for empty slots in an existing match (backfilling). Matchmaking and match backfill requests are handed to a matchmaker for processing.

- **FlexMatch matchmaker to evaluate players and create matches.** A matchmaker builds the best possible player matches from the requests it receives. It has a rule set that defines a match's team structure and sets the criteria to use when evaluating players for a match. A game can have multiple matchmakers, with each building a different type of match.

- **Game session queue to place new matches.** A game session queue finds available computing resources to host a match. It determines where (in what regions) to look for resources and how to select the best available host for each match.

The following sections detail how matchmaking proceeds to form new game matches or to find new players for existing game matches.

### Matchmaking Process

Here's how requests for a new game match are handled with FlexMatch. This description assumes that a client-side game service is initiating matchmaking requests and tracking the matchmaking ticket status.

1. **Request matchmaking.** Players take some action in your game that triggers matchmaking, such as clicking a "Join Now" button or a group of players forming a party. Your game initiates a matchmaking request, identifying which matchmaker to use and including one or more players to be matched. The request includes any player information, such as skill level or preferences, that the matchmaker requires to build matches. Each request gets a matchmaking ticket ID, which your game uses to track the request status and take action as needed.

2. **Discover potential matches.** All matchmaking tickets are passed to the specified matchmaker and placed in its ticket pool for processing. A ticket remains in a ticket pool until it is matched or until it reaches the matchmaker's maximum time limit.

   To find player matches for a regular (non-large) match, the matchmaker makes continual passes through the ticket pool. On each pass, the matchmaker starts with the oldest ticket in the pool and evaluates the other tickets against it to find the best possible matches. A matchmaker's rule set determines (1) how many teams to create for a match, (2) the number of players to assign to each
team, and (3) how to evaluate each prospective player. Rules might set requirements for individual players, teams, or matches. For example, a rule might require all matched players to have a certain talent, or it might require at least one player on a team to play a certain character. A commonly used rule requires that all players in a match have similar skill ratings.

For large matches, the process is slightly different. Instead of evaluating each player against a set of rules, FlexMatch weighs available matchmaking tickets against a single key balancing attribute and groups players that have similar attribute values. It also applies latency requirements. You can choose between either creating larger matches with greater similarity between players, or placing players into matches that give them the best possible latency experience. Once an initial match is found, FlexMatch then does a series of tests to ensure that the final match is the best available solution.

When the matchmaker evaluates a ticket, it either passes or fails the entire ticket. For tickets with multiple players, the matchmaker assumes these players want to play together and attempts to place them all in the same match. This means that, for any potential match, all the players in a ticket must be acceptable. If any player fails any rule, the entire ticket is considered not a match. Tickets that fail remain in the ticket pool and are evaluated again on the next pass. Once a potential match is filled, the status of all tickets in the match are updated.

3. **Get player acceptance.** If the matchmaker requires players to accept a potential match, FlexMatch cannot proceed with the match until every player accepts. The matchmaking ticket status is changed to indicate that acceptance is required, which prompts your game to request acceptances from all players in each matched ticket.

Players can choose to accept or reject a potential match. Your game collects the player responses and reports them back to FlexMatch. All players in the potential match must accept the match within a certain time limit to continue. If any player rejects the match or fails to respond before the time limit, the matchmaker drops the potential match. Tickets for players who accepted the match are returned to the matchmaker's ticket pool; tickets for players who did not accept the match move to a failure status and are no longer processed.

4. **Find resources to host the match.** Once a potential match is made and accepted, FlexMatch tries to place the match with available hosting resources. The matchmaker is configured to use a specific game session queue, and it passes the potential match to that queue for placement. The queue uses a set of rules to search regions and fleets for the best available server process to host the match. If the original matchmaking request contained player latency data, the queue uses this information to find resources that offer the lowest latency and most consistent gameplay experience for players in the match.

Once an available server process is located, Amazon GameLift creates a game session record with game properties and matchmaker data, including team structure and sizes, player assignments, and relevant player characteristics.

5. **Start a new game session.** As when starting any new game sessions, Amazon GameLift sends a start request to the server process along with the game session and matchmaker information. The server process takes the information and uses it to start a new game session for a matched game. When the game session is ready to accept players, the server process notifies Amazon GameLift.

6. **Connect players to the new game session.** Once the game session is ready for players, Amazon GameLift creates new player sessions for every player in the match. It then updates all matchmaking tickets, changing the ticket status to indicate success and adding connection information for all players. This change in ticket status, prompts your game to relay the connection information to game clients. Players can now join the game and claim their slots in the match and their team assignments.

### Backfill Process

Here's how finding new players for an existing match is handled with FlexMatch. Since match backfill requires up-to-date information on player slot availability in game sessions, we recommend initiating match backfill requests from the game server. Another option is to use a client-side game service, such as
a session directory service, that tracks game session and player activity. See more on adding the match
backfill feature to your game at Backfill Existing Games with FlexMatch (p. 76).

If a matchmaking configuration has automatic backfill enabled, the process is similar. The only difference
is that the initial backfill request is generated by GameLift instead of by your code. Automatic backfill
requests are triggered when a game session has an open player slot.

1. **Request backfill matchmaking.** A matched game has empty player slots that need to be filled. Your
game initiates a backfill request, identifying which matchmaker to use and describing the current
players in the game session. Each request has a matchmaking ticket ID, which your game uses to track
the request status and take action as needed. With automatic backfill, this ticket ID is added to the
game session's matchmaking data.

2. **Discover potential matches.** Matchmaking tickets for backfills are passed to the specified
matchmaker and placed in the same pool as tickets for new matches. For large matches only, backfill
tickets are prioritized over tickets for new matches.

   The matchmaker evaluates tickets and players equally, whether the tickets are for new players
or a backfill request. The one exception is that a potential match cannot have more than one
backfill ticket. A backfill ticket must be matched with at least one other ticket in order to complete
successfully, even when the matchmaker's rules allow a match to complete with empty player slots.
Once a potential match is filled, the status of all tickets in the match is updated.

3. **Get player acceptance.** If acceptance is required, only the new players need to accept a backfill match,
and this step is handled as described for matchmaking requests. The current players do not need to
accept a match that they're already playing in. As a result, even though the backfill request's ticket
status indicates that acceptance is required, your game does not need to take action.

   If any of the proposed new players fails to accept the match within the time limit, the potential match
is dropped and no new players are added to the existing match. When this happens, the ticket for the
backfill request returns to the ticket pool for processing.

4. **Update existing game session with new match data.** When a backfill match is successfully made
there is no need to place a new game session. Instead, Amazon GameLift updates the match data for
the existing game session, adding the new players and team assignments. Amazon GameLift sends the
updated game session information to the server process that is hosting the existing game.

5. **Connect new players to the existing game session.** Amazon GameLift creates player sessions for the
new players and updates the matchmaking tickets with current status, player sessions, and connection
information. Your client game service, which is tracking ticket status of the new players, relays the
connection information to the game clients. Players can now join the existing game and claim their
player slot.

---

**Game Architecture with Managed Amazon GameLift**

The diagram shown below illustrates the key components of a game architecture that is hosted using the
managed GameLift solution.
Key components are described as follows.

**Game clients**

To join a game being hosted on GameLift, your game client must first find an available game session. The game client searches for existing game sessions, requests matchmaking, or starts a new game session by communicating with GameLift service. This communication is done through a backend client service in order to help game owners maintain secure control of their game servers and hosting resources. The client service makes requests to the GameLift service and in response receives game session information, including connection details, which it relays it back to the game client. The game client then uses this information to connect directly to the game server and join the game. The green arrow represents the direct connection between game client and game server during gameplay.

**Client services**

A backend client service handles communication between game clients and the GameLift service by calling the GameLift service APIs in the AWS SDK. Client services might also be used for other game-specific tasks such as player authentication and authorization, inventory, or currency control. For example, when a player joins a game, your game client might first call an authentication service to first verify the player's identity, and only then send a player slot request to the GameLift service. Relevant information, such as connection details, are relayed back to the game client.

**External services**
Your game may rely on an external service, such as for validating a subscription membership. As shown in the architecture diagram, information from an external service can be passed to your game servers (via a client service and the GameLift service) without going through the game client.

**Game servers**

Your game server software is uploaded to the GameLift service and is deployed onto hosting machines to host game sessions and accept player connections. Game servers communicate with the GameLift service by using the GameLift Server SDK, exchanging requests to start new game sessions, validate newly connected players, and report status of game sessions, player connections, and available resources. Game clients connect directly to a game server after receiving connection details from the GameLift service.

**GameLift service**

The GameLift service is the core service that deploys and manages fleets of resources to host your game servers, coordinates how game sessions are placed across your available resources, starts and stops game sessions, and tracks game server health and activity in order to maintain game availability as player traffic fluctuates. When setting up and managing hosting resources, game owners use the GameLift service APIs in the AWS SDK and CLI to upload game server builds, create and configure fleets, and manage fleet capacity. Client services start new game sessions, request matchmaking, and slot players into game sessions by calling the GameLift service APIs in the AWS SDK. Game servers that are deployed onto GameLift fleets use the GameLift Server SDK to maintain communication with the GameLift service to start and stop game sessions, report server health, exchange game and player data as needed, etc.

**Hosting management tools**

The GameLift tool set in the AWS SDK provides multiple ways for you to configure your game hosting resources, scale capacity to player demand, and monitor the current status of resources, as well as track metrics on game server performance and game and player activity. In addition, you can remotely access any individual game server for troubleshooting.

---

**Game Architecture with Realtime Servers**

The diagram shown below illustrates the key components of a game architecture that is hosted using the managed GameLift with Realtime Servers solution.
Key components are defined as follows:

Game clients

To join a game being hosted on GameLift, your game client must first find an available game session. The game client searches for existing game sessions, requests matchmaking, or starts a new game session by communicating with GameLift service. This communication is done through a backend client service in order to help game owners maintain secure control of their game servers and hosting resources. The client service makes requests to the GameLift service and in response receives game session information, including connection details, which the client service relays back to the game client. The game client then uses this information, with the Realtime Client SDK, to connect directly to the game server. Once connected, the game client can join the game and exchange game state updates with other players in the game. The green arrow represents the direct connection between game client and game server during gameplay.

Client services

A backend client service handles communication between game clients and the GameLift service by calling the GameLift service APIs in the AWS SDK. Client services might also be used for other game-specific tasks such as player authentication and authorization, inventory, or currency control. For example, when a player joins a game, your game client might first call an authentication client service to verify the player's identity, and only then send a game session request to the GameLift service. Relevant information that the client service receives from the GameLift service, such as connection details, are relayed back to the game client.
External services

Your game may rely on an external service, such as for validating a subscription membership. As shown in the architecture diagram, information from an external service can be passed to your game servers (via a client service and the GameLift service) without going through the game client.

Realtime Servers

To host game sessions, you create a Realtime Servers fleet that is configured for your game. The Realtime servers take the place of an integrated full-fledged game server; instead they run script, which you customize for your game and upload to GameLift. Realtime servers track player connections to a game session and relay game data between players to keep each player’s game state in sync. They also communicate with the GameLift service to start new game sessions, validate newly connected players, and report on the status of game sessions, player connections, and available resources. When joining a game, a game client connects directly to a Realtime server after receiving connection details from the GameLift service.

GameLift service

The GameLift service is the core service that deploys and manages fleets of resources to host your Realtime servers, coordinates how game sessions are placed across your available resources, starts and stops game sessions, and tracks game server health and activity in order to maintain game availability as player traffic fluctuates. When setting up and managing hosting resources, game owners use the GameLift service APIs in the AWS SDK and CLI to upload game server builds and scripts, create and configure fleets, and manage fleet capacity. Client services start new game sessions, request matchmaking, and slot players into game sessions by calling the GameLift service APIs in the AWS SDK.

Hosting management tools

The GameLift tool set in the AWS SDK provides multiple ways for you to configure your game hosting resources, scale capacity to player demand, and monitor the current status of resources, as well as track metrics on game server performance and game and player activity. In addition, you can remotely access any individual game server for troubleshooting.
Setting Up

The topics in this section provide help with setting up your AWS account for the GameLift service and getting the necessary tools to use GameLift to host your multiplayer games.

Tip
Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using sample games (p. 29).

Topics
- Set Up an AWS Account (p. 21)
- Set Up a Role for Amazon GameLift Access (p. 22)
- Amazon GameLift SDKs (p. 23)
- Tools and Resources (p. 26)
- Billing Alerts (p. 27)
- Using Amazon GameLift in AWS Regions (p. 27)

Set Up an AWS Account

Amazon GameLift is an AWS service, and you must have an AWS account to use Amazon GameLift. Creating an AWS account is free.

For more information on what you can do with an AWS account, see Getting Started with AWS.

Set up your account for Amazon 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 into an existing one.

2. Set up user groups and access permissions. Open the AWS Identity and Access Management (IAM) service console and follow these steps to define a set of users or user groups and assign access permissions to them. Permissions are extended to a user or user group by attaching an IAM policy, which specifies the set of AWS services and actions a user should have access to. For detailed instructions on using the Console (or the AWS CLI or other tools) to set up your user groups, see Creating IAM Users.
   a. Create an administrative user or user group. Administrative users include anyone who manages core Amazon GameLift resources, such as builds and fleets. To set permissions, you must create your own policy from scratch. This example (p. 21) illustrates an administrator policy for Amazon GameLift services.
   b. Create a player user. A player user represents your game client(s). It enables access to Amazon GameLift client functionality, such as acquiring game session information and joining players to games. Your game client must use the player user credentials when communicating with the Amazon GameLift service. To set permissions, you must create your own policy from scratch. This example (p. 21) illustrates a player policy for Amazon GameLift services.

IAM Policy Examples for Amazon GameLift

You can use the following examples to create policies and add the appropriate permissions to your IAM users or user groups.
Simple Policy for Administrators

This policy provides full administrative access to a user. Attach it to a user or user group to permit all Amazon GameLift actions on all Amazon GameLift resources (fleets, aliases, game sessions, player sessions, etc.).

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

Simple Policy for Players

This policy enables access only to functionality needed by players who are using a game client to connect to an Amazon GameLift-hosted game server. This policy allows a user to get game session information, create new game sessions, and join a game session.

```
{
    "Version": "2012-10-17",
    "Statement":
    {
        "Effect": "Allow",
        "Action": [ "gamelift:CreateGameSession", "gamelift:DescribeGameSessions",
                    "gamelift:SearchGameSessions", "gamelift:CreatePlayerSession" ],
        "Resource": "*"
    }
}
```

Set Up a Role for Amazon GameLift Access

This topic refers to use of the managed GameLift solutions. If you're using GameLift FleetIQ as a standalone feature, see Setting up your AWS account for GameLift FleetIQ (p. 202).

Some GameLift features require you to extend limited access to your AWS resources. This is done by creating an AWS Identity and Access Management (IAM) role. A role specifies two things: (1) who can assume the role, and (2) which resources they can control while using the role. This topic provides guidance on how to set up a role to extend access to the Amazon GameLift service.

To set up an IAM role for the GameLift service

By creating a role specifically for GameLift, you define which of your AWS resources can be accessed either by the GameLift service or by your applications (such as game servers) that are running on GameLift.

Currently, a service-specific role for Amazon GameLift must be manually constructed with inline permissions and trust policies. You can update the role any time using the IAM service via the console or the AWS CLI.

1. Create an IAM role by using the AWS CLI. (The IAM console currently does not allow you to create a generic service role and add or edit policies.) See the IAM user guide topic Creating a Role for a
Service (AWS CLI) for specific instructions. The role must be created under the AWS account that you use to manage GameLift, so make sure you're using the proper AWS account credentials.

2. Create an inline permissions policy and attach it to the role. The permissions policy is where you specify what level of access that is covered by the role. You can specify access to a service or a resource, such as an Amazon S3 bucket, and you can limit permissions to specific actions. You can opt to create separate IAM roles for different sets of permissions to limit vulnerability. See these Examples of Policies for Delegating Access. Once you've defined your policy syntax, attach it to the service as described in the instructions linked to in Step 1.

3. Create a trust policy and attach it to the role. A trust policy specifies which AWS service(s) can assume the role. Use the following syntax:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "gamelift.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

4. Once you've created the role, you can view it in the IAM console. Make a note of the new role's ARN, as you may need to use it when setting up a GameLift feature.

Amazon GameLift SDKs

This topic describes the SDKs for the managed GameLift solution and the managed GameLift solution with Realtime Servers. To learn more about other GameLift solutions, see What Is Amazon GameLift? (p. 1).

Use GameLift software development kits (SDKs) to develop GameLift-enabled multiplayer game servers, game clients and game services that need to communicate with the GameLift service.

For detailed information on using the GameLift SDKs with your game engine, see Game Engines and Amazon GameLift (p. 34).

For Custom Game Servers

Create and deploy 64-bit custom game servers with the GameLift Server SDK. This SDK enables the GameLift service to deploy and manage game server processes across your GameLift hosting resources. Download the Server SDK and learn about how to Add Amazon GameLift to Your Game Server (p. 42) projects. See the GameLift Release Notes for version-specific information.

SDK support

The GameLift Server SDK download contains source for the following versions. Build the version you need for your game; see the README files with each version for build instructions and minimum requirements.

- C++
- C++ for Unreal Engine (plugin)
- C# (.NET)
Development environments

Build the SDK from source as needed for these supported development operating systems and game engines.

- **Operating systems** – Windows, Linux
- **Game engines** – Amazon Lumberyard, Unreal Engine, Unity, engines that support C++ or C# libraries

Game server operating systems

Use the GameLift Server SDK to create game servers that run on the following platforms:

- Windows Server 2012 R2
- Amazon Linux
- Amazon Linux 2

For Realtime Servers

Configure and deploy Realtime servers to host your multiplayer games, and enable your game clients to connect to them with the GameLift Realtime Client SDK. Game clients use this SDK to exchange messages with a Realtime server and with other game clients that are connected to the server. Download the Realtime Client SDK and learn about how to use it with your game clients (p. 59).

SDK support

The Realtime Client SDK contains source for the following languages:

- C# (.NET)

Development environments

Build the SDK from source as needed for these supported development operating systems and game engines.

- **Operating systems** – Windows, Linux, Android, iOS.
- **Game engines** – Unity, engines that support C# libraries

Game server operating systems

Realtime servers are deployed onto hosting resources that run the following platforms:

- Amazon Linux
- Amazon Linux 2

For Client Services

Create 64-bit client services using the AWS SDK with the GameLift API. This SDK enables client services to find or create game sessions and join players to games that are being hosted on GameLift. Download the AWS SDK or view the GameLift API reference documentation.

SDK support

The AWS SDK with Amazon GameLift is available in the following languages. See documentation for each language for details on support for development environments.
SDK Compatibility

The release history of the GameLift SDKs is as follows. There is no requirement to use comparable SDKs for your game server and client integrations, however older versions may not fully support use of the latest features.

<table>
<thead>
<tr>
<th>Release</th>
<th>AWS SDK version:</th>
<th>Server SDK version:</th>
<th>Realtime Client SDK version:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-10-24</td>
<td>1.7.210 (commit) or later</td>
<td>3.4.0</td>
<td>1.1.0</td>
</tr>
<tr>
<td>2019-09-03</td>
<td>1.7.175 (commit) or later</td>
<td>3.4.0</td>
<td>1.1.0</td>
</tr>
<tr>
<td>2019-07-09</td>
<td>1.7.140 (commit) or later</td>
<td>3.3.0</td>
<td>1.0.0</td>
</tr>
<tr>
<td>2019-04-25</td>
<td>1.3.58 (commit) or later</td>
<td>3.3.0</td>
<td>1.0.0</td>
</tr>
<tr>
<td>2018-12-14</td>
<td>1.3.58 (commit) or later</td>
<td>3.3.0</td>
<td></td>
</tr>
<tr>
<td>2018-02-15</td>
<td>1.3.58 (commit) or later</td>
<td>3.2.1</td>
<td></td>
</tr>
<tr>
<td>2018-02-08</td>
<td>1.3.52 (commit) or later</td>
<td>3.2.0</td>
<td></td>
</tr>
<tr>
<td>2017-08-16</td>
<td>1.1.31 (commit) or later</td>
<td>3.1.7</td>
<td></td>
</tr>
<tr>
<td>2017-02-21</td>
<td>1.0.72 (commit) or later</td>
<td>3.1.5</td>
<td></td>
</tr>
<tr>
<td>2016-09-01</td>
<td>0.14.9 (commit) or later</td>
<td>3.1.0 (C++ only)</td>
<td></td>
</tr>
<tr>
<td>2016-08-04</td>
<td>0.12.16 (commit) or later</td>
<td>3.0.7 (C++ only)</td>
<td></td>
</tr>
</tbody>
</table>

(Version information for the AWS SDK for C++ can be found in this file: aws-sdk-cpp/aws-cpp-sdk-core/include/aws/core/VersionConfig.h.)

For Amazon Lumberyard users, the following table lists the GameLift SDK versions that are bundled into or are compatible with the Lumberyard game engine.

<table>
<thead>
<tr>
<th>Amazon Lumberyard version:</th>
<th>Are bundled with GameLift SDK versions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4 to 1.5 (beta)</td>
<td>• Server SDK: 3.0.7</td>
</tr>
<tr>
<td></td>
<td>• AWS SDK: 0.12.16</td>
</tr>
</tbody>
</table>
Tools and Resources

Amazon GameLift provides a collection of tools and resources for you to use.

Core Tools

Use these tools to work with Amazon GameLift.

Amazon GameLift SDKs

The Amazon GameLift SDKs contain the libraries needed to communicate with the Amazon GameLift service from your game clients, game servers and game services. Versions of these SDKs are available with Lumberyard or you can download the latest versions separately. See details in Amazon GameLift SDKs (p. 23).

Realtime Client SDK for Amazon GameLift

For games using Realtime Servers, The Realtime Client SDK enables a game client to connect to a deployed Realtime server, join a game session, and keep its game state in sync with other players in the game. Download the SDK and learn more about making API calls with the Realtime Servers Client API (C#) Reference (p. 217).

AWS console for Amazon GameLift

Use the AWS Management Console for Amazon GameLift to manage your game deployments, configure resources, and track player usage and performance metrics. The Amazon GameLift console provides a GUI alternative to managing resources programmatically with the AWS SDK.

AWS CLI for Amazon GameLift

Use this command line tool to make calls to the AWS SDK, including the Amazon GameLift API. Get the AWS Command Line Interface download and install it using these instructions. You can also view the complete AWS CLI Command Reference for all AWS services, including Amazon GameLift.

Amazon GameLift Local

This client-side debugging tool emulates a subset of the Amazon GameLift API on your local development machine. You can test iterative code changes without needing to upload and run your game server on Amazon GameLift instances. Amazon GameLift Local can be used on Windows and Linux devices to test game clients and servers that use the Amazon GameLift SDKs. Amazon GameLift Local is available in the Server SDK download. See details in Testing Your Integration (p. 54).

Additional Resources

Use these resources to learn and experiment with Amazon GameLift for your multiplayer games.
Five-click sample

Get a sample multiplayer game (client and custom game server) up and running on Amazon GameLift in under an hour. This sample gives you a jump start so you can quickly start experimenting with Amazon GameLift tools, set up fleets and deploy the game server for hosting, adjust autoscaling, and track game activity and performance. You can find the sample in the Amazon GameLift console; from the GameLift intro page click "Test Amazon GameLift", or from any other console page select "Sample game" in the Amazon GameLift navigation menu.

Amazon GameLift forum

Use the Amazon GameLift forum to exchange ideas and knowledge, pick up tips, and get help with issues.

GameTech blog

Watch the game development blog to keep up with new features about Amazon GameLift, Amazon Lumberyard, learn more about game development with AWS, and get expert tips from the teams.

AWS Samples on GitHub

Find a large collection of code samples for all AWS services posted in AWS Samples.

Getting Started tutorials

Use the tutorials to walk through the process of getting a sample multiplayer game up and running on Amazon GameLift. Once you complete the series, you can use the game to explore other Amazon GameLift features and tools, such as auto-scaling and performance metrics.

Amazon GameLift product information

Check these pages for detailed descriptions of Amazon GameLift, including service overview, FAQ, and pricing details.

Amazon Lumberyard game engine

Amazon Lumberyard comes with the Amazon GameLift SDKs built in, and integration is handled automatically. It is bundled with a sample multiplayer game that illustrates how to use Lumberyard to integrate a game with Amazon GameLift. Find more information about this sample project in the Lumberyard User Guide.

Billing Alerts

You may want to set up and configure a billing alert to notify you of billing events. For more information, see Creating a Billing Alarm.

In addition to receiving billing alerts, you can view your current estimated bill for Amazon GameLift on the Billing and Cost Management console at https://console.aws.amazon.com/billing/. This will help you review your resource consumption, make decisions about future usage, and determine your scaling needs.

To avoid incurring unnecessary charges, you may want to scale down your fleet (p. 122) when not in use.

Using Amazon GameLift in AWS Regions

Amazon GameLift is available in multiple AWS regions. For a complete list of AWS regions and endpoints, see AWS Regions and Endpoints. A global AWS account allows you to work with resources in most regions.
When using Amazon GameLift for resources in the China (Beijing) Region, which is operated by Sinnet, keep in mind that you need to have a separate AWS (China) account. In addition, there are some differences in how Amazon GameLift is implemented, including how it handles cross-region interactions. For more information on using Amazon GameLift in the China (Beijing) Region, see these resources:

- AWS in China
- Amazon GameLift in China
Getting Started with Amazon GameLift

The resources in this section help you get started with the managed GameLift solutions for custom game servers and Realtime Servers.

Topics

- Explore Amazon GameLift (p. 29)
- Get Started with Custom Servers (p. 30)
- Get Started with Realtime Servers (p. 31)

Explore Amazon GameLift

Looking to experiment with Amazon GameLift features before diving in with your own game? Try out these sample experiences. The console samples give you hands-on experience with game hosting in the GameLift console. The source code example and walkthrough shows you how to prepare a game for hosting using Realtime Servers.

Realtime Servers Sample Game (Full Source)

Mega Frog Race is a complete multiplayer game sample with source code. Follow a hands-on tutorial that walks through how to prepare the sample to run online using GameLift Realtime Servers. This sample is a good way to better understand how to get your game client ready to work with GameLift Realtime Servers. You can also use it as a starting point to experiment with other GameLift features such as FlexMatch.

To read the hands-on tutorial, see the GameTech blog post Creating Servers for Multiplayer Mobile Games with Just a Few Lines of JavaScript.

To get the source code, go to the GitHub repository.

The source material for the MegaFrogRace sample includes all the elements to deploy the multiplayer game hosted with Realtime Servers.

- Game client – Source code for the Unity-created C++ game client. It illustrates how to get game session connection information from GameLift, connect to a Realtime server, and exchange game updates with other players via the Realtime server.
- Client service – Source (in Node-based JavaScript) for an AWS Lambda function that manages direct API calls to the GameLift service. When called by the game client, the service makes requests to find or start new game sessions and assign players, and then returns connection details back to the game client.
- Realtime script – Source script file (in Node-based JavaScript) that configures a fleet of Realtime servers for the game. This script includes the minimum configuration to enable Realtime servers to communicate with the GameLift service and to start and stop game sessions. It also includes some custom logic for the sample game.
Custom Game Server Sample (Console Experience)

This sample experience quickly gets you working with a live game on GameLift. Upload a sample game build, create a fleet to run the game server, and connect to it from a sample game client. You can start up multiple game clients and play to generate hosting data. Once you have some data, explore the GameLift console to view your hosting resources, track metrics, and experiment with ways to scale hosting capacity.

To access the sample wizard, sign into the GameLift console, open the Amazon GameLift menu, and select Custom game server sample.

About the sample game

The sample game is developed using the Amazon Lumberyard game engine. To run the game client, you need a Windows 7 64-bit system and 300 MB of space. See additional requirements.

Get Started with Custom Servers

This roadmap outlines the key steps to getting your multiplayer games with custom game servers up and running with the managed GameLift solution for custom game servers. If you're interested in using GameLift Realtime Servers, which lets you deploy your game client with our ready-to-deploy game servers, see Get Started with Realtime Servers (p. 31). To learn about other GameLift solutions, see What Is Amazon GameLift? (p. 1).

Tip

Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using sample games (p. 29).

New to GameLift? We recommend that you read What Is Amazon GameLift? (p. 1) If you're unsure whether GameLift supports your operating systems and development environments, see the topics Amazon GameLift SDKs (p. 23) and Game Engines and Amazon GameLift (p. 34).

Before you start integration, you need to have an AWS account and configure it for Amazon GameLift. Learn more at Set Up an AWS Account (p. 21). All essential tasks related to creating and managing your game servers can be done using the Amazon GameLift console, but you may also want to Get and install the AWS Command Line Interface tool.

1. Prepare your custom game server for hosting on Amazon GameLift.
   • Get the Amazon GameLift Server SDK and build it for your preferred programming language and game engine. If you're using the Amazon Lumberyard game engine, a version of the SDK is built in. See the Amazon GameLift SDKs For Custom Game Servers (p. 23) and Game Engines and Amazon GameLift (p. 34).
   • Add code to your game server project to enable communication with the Amazon GameLift service. A game server must be able to notify Amazon GameLift about its status, start/stop game sessions when prompted, and other tasks. See Add Amazon GameLift to Your Game Server (p. 42).

2. Prepare your game client to connect to Amazon GameLift-hosted game sessions.
   • Set up a client service to communicate with Amazon GameLift service in order to start game sessions and place players into games when prompted by a game client.
     • Add the AWS SDK to your client service project. See the Amazon GameLift SDKs For Client Services (p. 24).
     • Add functionality to retrieve information on game sessions, place new game sessions and (optionally) reserve space for players on a game session. See Add Amazon GameLift to Your Game Client (p. 47). Recommended: Use game session placements to take advantage of
FleetIQ and optimize resource usage and player experience. This option is required if you’re using FlexMatch.

- (optional) Enable the client service to request player matchmaking using FlexMatch. Learn more in FlexMatch Integration Roadmap (p. 70).

- Enable your game client to connect directly with a hosted game session. Add code to acquire connection information for a game session and (optionally) a reserved player session. Use this connection information and a unique player ID to communicate with the game server and join the game. See Join a Player to a Game Session (p. 50).

3. Test your Amazon GameLift integration.

- Use Amazon GameLift Local to test your game client and game server integration using a version of the Amazon GameLift service running locally. You can use this tool to test your integration without having to upload game builds and set up fleets. You can verify that your game components are communicating with the Amazon GameLift service, and test core functionality. See Testing Your Integration (p. 54).

4. Build a fleet of computing resources to host your game.

- Package and upload your custom game server build to the Amazon GameLift service. Be sure to upload your build to each region where you plan to deploy your game. See Upload a Custom Server Build to GameLift (p. 91).

- Design a fleet configuration for your game. Decide, for example, the type of computing resources to use, which regions to deploy to, whether to use queues, and other options. See Design a Amazon GameLift Fleet for Your Game (p. 99).

- Create fleets and deploy them with your custom game server. Once a fleet is active, it is ready to host game sessions and accept players. See Setting Up Amazon GameLift Fleets (p. 99).

- Experiment with Amazon GameLift fleet configuration settings and refine as needed to optimize usage of your fleet resources. Adjust the number of game sessions to run concurrently on each instance, or set game session activation limits. See Design a Amazon GameLift Fleet for Your Game (p. 99). See also how to Remotely Access Fleet Instances (p. 117).

- Create a queue to manage how new game sessions are placed with available hosting resources. See Design a Game Session Queue (p. 128).

- Enable auto-scaling to manage your fleet’s hosting capacity for expected player demand. See Scaling Amazon GameLift Fleet Capacity (p. 119).

- (optional) Set up a FlexMatch matchmaker with a set of custom matchmaking rules for your game. Learn more in FlexMatch Integration Roadmap (p. 70).

Note
Once you’ve created your queues, you’ll need to update your client service to use the correct queue ID when request game session placements and/or matchmaking.

Once you've fully integrated Amazon GameLift into your game components, it's a matter of managing your game server fleets for optimal availability and performance over the long term. Use Amazon GameLift tools to track things like how quickly and efficiently players can find and connect to a game session, overall performance of your game servers over time, and player usage patterns. See Viewing Your Game Data in the Console (p. 170).

Get Started with Realtime Servers

This roadmap outlines the key steps to getting your multiplayer game clients up and running with the managed GameLift solution with Realtime Servers. If you have a game with a custom game server, see Get Started with Custom Servers (p. 30). To learn about other GameLift solutions, see What Is Amazon GameLift? (p. 1).
New to Realtime Servers or unsure about whether this feature is appropriate for your game? We recommend that you read How Realtime Servers Work (p. 8).

Note
If you're familiar with how to integrate and deploy games with GameLift, here's a quick summary of what's different with Realtime Servers:

- Create and upload a Realtime script with optional game logic to run game sessions on Realtime Servers instances. You no longer need to develop a custom game server and integrate it with the GameLift Server SDK.
- When creating a fleet to host your game sessions, deploy it with the Realtime script instead of a game server build.
- Integrate your game client with the Realtime Client SDK to manage connections to game sessions.

Before you start integration, you need to have an AWS account and configure it for GameLift. Learn more at Set Up an AWS Account (p. 21). All essential tasks related to creating and managing your game servers can be done using the GameLift console, but you may also want to Get and install the AWS Command Line Interface tool.

1. **Create a Realtime script for hosting on GameLift.**
   - Create a Realtime script with your server configuration and optional custom game logic. Realtime Servers are already built to start and stop game sessions, accept player connections, and manage communication with the GameLift service and between players in a game. There are also hooks that allows you to add custom server logic for your game. Realtime Servers is based on Node.js, and server script is written in JavaScript. See Creating a Realtime Script (p. 63).

2. **Build a fleet of computing resources to host your game.**
   - Upload the Realtime script to the GameLift service. Be sure to upload your script to each region where you plan to deploy your game. See Upload a Realtime Servers Script to Amazon GameLift (p. 96).
   - Design a fleet configuration for your game. Decide, for example, the type of computing resources to use, which regions to deploy to, whether to use queues, and other options. See Design an Amazon GameLift Fleet for Your Game (p. 99).
   - Create Realtime Servers fleets and deploy them with the Realtime script. Once a fleet is active, it is ready to host game sessions and accept players. See Setting Up Amazon GameLift Fleets (p. 99).
   - Experiment with GameLift fleet configuration settings and refine as needed to optimize usage of your fleet resources. Adjust the number of game sessions to run concurrently on each instance, or set game session activation limits. See Design an Amazon GameLift Fleet for Your Game (p. 99). See also how to Remotely Access Fleet Instances (p. 117).
   - Create a queue to manage how new game sessions are placed with available hosting resources. See Design a Game Session Queue (p. 128).
   - Enable auto-scaling to manage your fleet's hosting capacity for expected player demand. See Scaling Amazon GameLift Fleet Capacity (p. 119).
   - (optional) Set up a FlexMatch matchmaker with a set of custom matchmaking rules for your game. Learn more in FlexMatch Integration Roadmap (p. 70).

3. **Prepare your game client to join GameLift-hosted game sessions.**
   - Create a mechanism to assign unique player IDs for use with GameLift.
   - Set up a client service to send requests to the GameLift for new game sessions and to reserve space for players in existing game sessions. See Add Amazon GameLift to Your Game Client (p. 47).
• (optional) Enable the client service to request player matchmaking using FlexMatch. Learn more in FlexMatch Integration Roadmap (p. 70).

• Enable your game client to connect directly with a hosted game session that is running on a Realtime server and exchange information through messaging. See Integrating a Game Client for Realtime Servers (p. 59).

Once you've fully integrated GameLift and Realtime Servers into your game components, it's a matter of managing your game server fleets for optimal availability and performance over the long term. Use GameLift tools to track things like how quickly and efficiently players can find and connect to a game session, overall performance of your game servers over time, and player usage patterns. See Viewing Your Game Data in the Console (p. 170).
Preparing Games for Amazon GameLift

To get your multiplayer game up and running on the managed GameLift service, you need to do some work to set up communication between your game and the GameLift service. The guides in this section provide detailed help for integrating your game with GameLift, whether you plan to deploy a custom game server or use GameLift ready-built Realtime Servers, and for adding a matchmaking service using FlexMatch.

Tip
Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using sample games (p. 29).

Topics
• Integrating Games with Custom Game Servers (p. 34)
• Integrating Games with Amazon GameLift Realtime Servers (p. 59)
• Adding FlexMatch Matchmaking (p. 69)

Integrating Games with Custom Game Servers

Amazon GameLift provides a full tool set for preparing your multiplayer games and custom game servers to run on the managed GameLift service. The GameLift SDKs contain libraries needed to enable game clients and servers to communicate with the GameLift service. The SDKs can be downloaded directly; they are also included in the Amazon Lumberyard game engine. For more details on the SDKs and where to get them, see Amazon GameLift SDKs (p. 23).

The topics in this section contain detailed instruction on how to add the necessary GameLift functionality to your game client and game server before deploying on GameLift. For a complete roadmap to getting your game up and running on GameLift, see Get Started with Custom Servers (p. 30).

Topics
• Game Engines and Amazon GameLift (p. 34)
• Integrating your Game Server for Amazon GameLift (p. 42)
• Integrating your Game Client for Amazon GameLift (p. 47)
• Amazon GameLift and Game Client/Server Interactions (p. 51)
• Testing Your Integration (p. 54)

Game Engines and Amazon GameLift

You can use the managed GameLift service with most major game engines that support C++ or C# libraries, including Amazon Lumberyard, Unreal Engine, and Unity. Build the version you need for your game; see the README files with each version for build instructions and minimum requirements. For more information on available GameLift SDKs, supported development platforms and operating systems, see Amazon GameLift SDKs (p. 23) for game servers.

In addition to the engine-specific information provided in this topic, find additional help with integrating GameLift into your game servers, clients and services in the following topics:

• Get Started with Custom Servers (p. 30) – A six-step workflow for successfully integrating GameLift into your game and setting up hosting resources.
- **Add Amazon GameLift to Your Game Server (p. 42)** – Detailed instructions on integrating GameLift into a game server.
- **Add Amazon GameLift to Your Game Client (p. 47)** – Detailed instructions on integrating into a game client or service, including creating game sessions and joining players to games.

**Amazon Lumberyard**

GameLift SDKs and functionality are fully incorporated into the Lumberyard product.

**Game servers**

Prepare your game servers for hosting on GameLift using the *GameLift Server SDK for C++* (p. 232). See **Add Amazon GameLift to Your Game Server (p. 42)** to get help with integrating the required functionality into your game server.

**Game clients and services**

Enable your game clients and/or game services to interact with GameLift service, such as to find available game sessions or create new ones, and add players to games. Core client functionality is provided in the *AWS SDK for C++*. To integrate GameLift into your Lumberyard game project, see **Prepare Your Game Client in Amazon Lumberyard (p. 36)** and **Add Amazon GameLift to Your Game Client (p. 47)**.

**Unreal Engine**

**Game servers**

Prepare your game servers for hosting on GameLift by adding the *GameLift Server SDK for Unreal Engine* (p. 261) to your project and implementing the required server functionality. For help setting up the Unreal Engine plugin and adding GameLift code, see **Add Amazon GameLift to an Unreal Engine Game Server Project (p. 36)**.

**Game clients and services**

Enable your game clients and/or game services to interact with GameLift service, such as to find available game sessions or create new ones, and add players to games. Core client functionality is provided in the *AWS SDK for C++*. To integrate GameLift into your Unreal Engine game project, see **Add Amazon GameLift to Your Game Client (p. 47)**.

**Unity**

**Game servers**

Prepare your game servers for hosting on GameLift by adding the *GameLift Server SDK for C#* (p. 248) to your project and implementing the required server functionality. For help setting up with Unity and adding GameLift code, see **Add Amazon GameLift to a Unity Game Server Project (p. 40)**.

**Game clients and services**

Enable your game clients and/or game services to interact with GameLift service, such as to find available game sessions or create new ones, and add players to games. Core client functionality is provided in the *AWS SDK for .NET*. To integrate GameLift into your Unity game project, see **Add Amazon GameLift to Your Game Client (p. 47)**.

**Other Engines**

For a full list of the GameLift SDKs available for game servers and clients, see the section called “GameLift SDKs” (p. 23).
Prepare Your Game Client in Amazon Lumberyard

All game clients must be configured to enable communication with the managed GameLift service, including specifics on which fleet to use, access credentials, how to connect, etc. The simplest method is to create a batch file that sets the console variables listed as follows.

Tip
Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using sample games (p. 29).

To prepare the game client

1. In your batch file, set the following console variables to launch the game client. These variables have been added to `<dev\Code\CryEngine\CryNetwork\Lobby\LobbyCvars`
   - `gamelift_aws_access_key` = part of the IAM security credentials (p. 21) for a user with "player" access in your AWS account
   - `gamelift_aws_secret_key` = part of the IAM security credentials (p. 21) for a user with "player" access in your AWS account
   - `gamelift_fleet_id` = Unique ID of an active fleet to connect to
   - `gamelift_alias_id` = Unique ID of an alias pointing to a fleet to connect to
   - (Optional) `gamelift_endpoint` = GameLift server endpoint; the default value is `gamelift.us-west-2.amazonaws.com`
   - (Optional) `gamelift_aws_region` = AWS region name; default value is `us-west-2`
   - (Optional) `gamelift_player_id` = ID that you generate to uniquely identify a player (p. 51)

2. Add the following command to launch the server browser:

   Follow this pattern when using an GameLift fleet ID (`gamelift_fleet_id`):

   ```
   Bin64\[your game executable] +gamelift_fleet_id [your fleet ID] +gamelift_aws_region us-west-2 +gamelift_aws_access_key [your AWS access key] +gamelift_aws_secret_key [your AWS secret key] +sv_port 64091 +map [map name]
   ```

   Follow this pattern when using an GameLift alias ID (`gamelift_alias_id`):

   ```
   Bin64\[your game executable] +gamelift_alias_id [your alias ID] +gamelift_aws_region us-west-2 +gamelift_aws_access_key [your AWS access key] +gamelift_aws_secret_key [your AWS secret key] +sv_port 64091 +map [map name]
   ```

Add Amazon GameLift to an Unreal Engine Game Server Project

This topic helps you set up and use the GameLift Server SDK plugin for Unreal Engine in your game server projects. If you're unsure whether the GameLift service supports the operating systems you're using, see For Custom Game Servers (p. 23).

Set Up the Unreal Engine Server SDK Plugin

Follow these steps to get the GameLift Server SDK plugin for Unreal Engine ready for your game server projects.

To set up the GameLift SDK plugin for Unreal Engine

1. Download the GameLift Server SDK. To verify that your game system requirements are supported, see Amazon GameLift SDKs (p. 23).
2. **Build the C++ Server SDK libraries for Unreal.** The SDK download contains the source code for C++ (see `GameLift-<release date>/GameLift-SDK-Release-<version>/GameLift-cpp-ServerSDK-<version>`). Check the README file in this directory for minimum requirements and additional information before building the SDK.

To build the SDK libraries, go to the directory `GameLift-cpp-ServerSDK-<version>` and compile with the flag `-DBUILD_FOR_UNREAL` set to true. The following instructions show how to compile using cmake.

For Linux users:

```bash
mkdir out
cd out
cmake -DBUILD_FOR_UNREAL=1 ..
make
```

The following binary files are generated:

- `out/prefix/lib/libaws-cpp-sdk-gamelift-server.so`

For Windows users:

```bash
mkdir out
cd out
cmake -G "Visual Studio 15 2017 Win64" -DBUILD_FOR_UNREAL=1 ..
msbuild ALL_BUILD.vcxproj /p:Configuration=Release
```

The following binary files are generated:

- `out\prefix\bin\aws-cpp-sdk-gamelift-server.dll`
- `out\prefix\lib\aws-cpp-sdk-gamelift-server.lib`

For more details on building the C++ SDK, including minimum requirements and build options, see the README.md file included in the download.

3. **Add the binaries to the GameLift plugin files.** Open the directory for the plugin version of UE4 that you are working with (for example, `GameLift-SDK-Release-3.3.0/GameLift-Unreal-plugin-3.3.0/UE4.21.1/GameLiftServerSDK`). Copy the binary files that you created in Step 2 into the ThirdParty directory of the Unreal plugin:

For Linux use these paths:

- `.../ThirdParty/GameLiftServerSDK/Linux/x86_64-unknown-linux-gnu/aws-cpp-sdk-gamelift-server.so`

for Windows use these paths:

- `...\ThirdParty\GameLiftServerSDK\Win64\aws-cpp-sdk-gamelift-server.dll`
- `...\ThirdParty\GameLiftServerSDK\Win64\aws-cpp-sdk-gamelift-server.lib`

4. **Import the GameLift plugin into a project.** There are many ways to import a plugin into Unreal Engine. The following method does not require the Unreal Editor.

a. Add the plugin to your game project. The plugin files must contain everything in the plugin's `GameLiftServerSDK` directory, including the generated binary files.

b. Add the plugin to your game's .uproject file:
"Plugins": [
  {"Name": "GameLiftServerSDK",
   "Enabled": true}
]

c. Add the plugin name as a dependency to your game's list of ModuleRules. The following example shows a sample list of module names with the GameLift plugin added to it.

```cpp
using UnrealBuildTool;
public class MyAwesomeGame : ModuleRules
{
  public MyAwesomeGame(TargetInfo Target)
  {
    PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject",
      "Engine", "InputCore", "GameLiftServerSDK" });
  }
}
```

### Add GameLift Code

For more information on adding GameLift functionality, see these topics:

- Add Amazon GameLift to Your Game Server (p. 42)
- Amazon GameLift Server API Reference for Unreal Engine (p. 261)

When adding GameLift-specific code to your Unreal Engine game project, enclose the code using the preprocessor flag `WITH_GAMELIFT=1`. This flag ensures that only server builds invoke the GameLift backplane API and allows you to write code that is executed correctly regardless of the build target type you might produce with it.

Code enclosed with the `WITH_GAMELIFT=1` flag is only processed if the following are true:

- The plugin found the GameLift server SDK binary files.
- The build is a game server: `Target.Type == TargetRules.TargetType.Server`

The following code snippet illustrates how to initialize an Unreal Engine game server with GameLift.

```cpp
//This is an example of a simple integration with GameLift server SDK that makes game server
//processes go active on Amazon GameLift

// Include game project files. "GameLiftFPS" is a sample game name, replace with file names from your own game project
#include "GameLiftFPSGameMode.h"
#include "GameLiftFPS.h"
#include "Engine.h"
#include "EngineGlobals.h"
#include "GameLiftFPSHUD.h"
#include "GameLiftFPSCharacter.h"
#include "GameLiftServerSDK.h"

AGameLiftFPSGameMode::AGameLiftFPSGameMode()
: Super()
```
// Let's run this code only if GAMELIFT is enabled. Only with Server targets!
#if WITH_GAMELIFT

// Getting the module first.
FGameLiftServerSDKModule* gameLiftSdkModule = &FModuleManager::LoadModuleChecked<FGameLiftServerSDKModule>(FName("GameLiftServerSDK"));

// InitSDK establishes a local connection with GameLift's agent to enable communication.
gameLiftSdkModule->InitSDK();

// Respond to new game session activation request. GameLift sends activation request
// to the game server along with a game session object containing game properties
// and other settings. Once the game server is ready to receive player connections,
// invoke GameLiftServerAPI.ActivateGameSession()
auto onGameSession = [=](Aws::GameLift::Server::Model::GameSession gameSession)
{
    gameLiftSdkModule->ActivateGameSession();
};

FProcessParameters* params = new FProcessParameters();
params->OnStartGameSession.BindLambda(onGameSession);

// OnOnProcessTerminate callback. GameLift invokes this before shutting down the instance
// that is hosting this game server to give it time to gracefully shut down on its own.
// In this example, we simply tell GameLift we are indeed going to shut down.
params->OnTerminate.BindLambda([=](){gameLiftSdkModule->ProcessEnding();});

// HealthCheck callback. GameLift invokes this callback about every 60 seconds. By
default,
// GameLift API automatically responds 'true'. A game can optionally perform checks on
// dependencies and such and report status based on this info. If no response is
// received
// within 60 seconds, health status is recorded as 'false'.
// In this example, we're always healthy!
params->OnHealthCheck.BindLambda([=](){return true; });

// Here, the game server tells GameLift what port it is listening on for incoming
// player
// connections. In this example, the port is hardcoded for simplicity. Since active game
// that are on the same instance must have unique ports, you may want to assign port
// values
// from a range, such as:
// const int32 port = FURL::UrlConfig.DefaultPort;
// params->port;
// params->port = 7777;

// Here, the game server tells GameLift what set of files to upload when the game
// session
// ends. GameLift uploads everything specified here for the developers to fetch later.
TArray<FString> logfiles;
logfiles.Add(TEXT("aLogFile.txt"));
params->logParameters = logfiles;

// Call ProcessReady to tell GameLift this game server is ready to receive game
// sessions:
gameLiftSdkModule->ProcessReady(*params);
#endif
Add Amazon GameLift to a Unity Game Server Project

This topic helps you set up the GameLift C# Server SDK in your Unity game server projects. If you're unsure whether the managed GameLift service supports the operating systems you're using, see For Custom Game Servers (p. 23).

Set up the C# Server SDK for Unity

Follow these steps to build the GameLift Server SDK for C# and add it to your Unity game server projects.

To set up the GameLift Server SDK for Unity

1. Download the GameLift Server SDK. To verify that your game system requirements are supported, see Amazon GameLift SDKs (p. 23).
2. Build the C# SDK libraries. See the README.md file for the C# Server SDK for minimum requirements and additional build options. In an IDE, load the solution file that you want to use. To generate the SDK libraries, restore the NuGet packages and build the solution.
3. Check the Configuration settings. In the Unity Editor, go to File, Build Settings, Player Settings. Under Other Settings, Configuration, check the following settings:
   - Scripting Runtime Version: Set to the .NET solution you're using.
4. Add the GameLift libraries to Unity. In the Unity Editor, import the libraries that were produced by the build into the Assets/Plugins directory of your project. See the README.md file for a complete list of libraries for the SDK version you're using.

Add GameLift Server Code

For more information on adding GameLift functionality, see these topics:

- Add Amazon GameLift to Your Game Server (p. 42)
- Amazon GameLift Server API (C#) Reference (p. 248)

The following code example uses a MonoBehaviour to illustrate a simple game server initialization with GameLift.

```csharp
using UnityEngine;
using Aws.GameLift.Server;
using System.Collections.Generic;

public class GameLiftServerExampleBehavior : MonoBehaviour
{
    // This is an example of a simple integration with GameLift server SDK that makes game server processes go active on Amazon GameLift
    public void Start()
    {
        // Set the port that your game service is listening on for incoming player connections (hard-coded here for simplicity)
        var listeningPort = 7777;

        // InitSDK establishes a local connection with the Amazon GameLift agent to enable further communication.
        var initSDKOutcome = GameLiftServerAPI.InitSDK();
        if (initSDKOutcome.Success)
        {
```

Version
40
ProcessParameters processParameters = new ProcessParameters(
    (gameSession) => {
        //Respond to new game session activation request. GameLift sends
        //activation request
        //to the game server along with a game session object containing game
        //properties
        //and other settings. Once the game server is ready to receive player
        //connections,
        //invoke GameLiftServerAPI.ActivateGameSession()
        GameLiftServerAPI.ActivateGameSession();
    },
    () => {
        //OnProcessTerminate callback. GameLift invokes this callback before
        //shutting down
        //an instance hosting this game server. It gives this game server a
        //chance to save
        //its state, communicate with services, etc., before being shut down.
        //In this case, we simply tell GameLift we are indeed going to shut
        //down.
        GameLiftServerAPI.ProcessEnding();
    },
    () => {
        //This is the HealthCheck callback.
        //GameLift invokes this callback every 60 seconds or so.
        //Here, a game server might want to check the health of dependencies
        //and such.
        //Simply return true if healthy, false otherwise.
        //The game server has 60 seconds to respond with its health status.
        //GameLift will default to 'false' if the game server doesn't respond
        //in time.
        //In this case, we're always healthy!
        return true;
    },
    () => {
        //Here, the game server tells GameLift what port it is listening on for
        //incoming player
        //connections. In this example, the port is hardcoded for simplicity.
        //Active game
        //that are on the same instance must have unique ports.
        //calling ProcessReady tells GameLift this game server is ready to receive
        //incoming game sessions!
        var processReadyOutcome = GameLiftServerAPI.ProcessReady(processParameters);
        if (processReadyOutcome.Success)
        {
            print("ProcessReady success.");
        }
        else
        {
            print("ProcessReady failure : " + processReadyOutcome.Error.ToString());
        }
    }
);

//Calling ProcessReady tells GameLift this game server is ready to receive
//incoming game sessions!
var processReadyOutcome = GameLiftServerAPI.ProcessReady(processParameters);
if (processReadyOutcome.Success)
{
    print("ProcessReady success.");
}
else
{
    print("ProcessReady failure : " + processReadyOutcome.Error.ToString());
}
else
{
    print("InitSDK failure : " + initSDKOutcome.Error.ToString());
}
void OnApplicationQuit()
{
    // Make sure to call GameLiftServerAPI.Destroy() when the application quits.
    // This resets the local connection with GameLift's agent.
    GameLiftServerAPI.Destroy();
}

Integrating your Game Server for Amazon GameLift

The topics in this section describe how to integrate managed GameLift into a multiplayer game server.

Adding GameLift to your game server is Step 2 on the Get Started with Custom Servers (p. 30) roadmap. The integration topics in this section assume that you've created an AWS account and have an existing game server project.

Topics

- Add Amazon GameLift to Your Game Server (p. 42)
- Amazon GameLift Server API (C++) Reference (p. 232)
- Amazon GameLift Server API (C#) Reference (p. 248)
- Amazon GameLift Server API Reference for Unreal Engine (p. 261)

Add Amazon GameLift to Your Game Server

Your custom game server needs to communicate with the managed GameLift service once it is deployed and running on GameLift instances. Each game server process must be able to respond to events when they are triggered by the GameLift service, and it must keep GameLift informed about the server process status and (optionally) player connections. For more details, see Amazon GameLift and Game Client/Server Interactions (p. 51).

Integrate the GameLift Server SDK to your game server. See Amazon GameLift SDKs (p. 23) to learn more about the Server SDK and download the latest version. The Server SDK is available in several languages, see these API references for complete information:

- C++ Server API Reference (p. 232)
- C# Server API Reference (p. 248)
- Unreal Engine Plugin API Reference (p. 261)

To integrate GameLift into your game server, add the GameLift Server SDK to your game server project and build the basic functionality described in this topic.

Prepare a Server Process

Add code to initialize an GameLift client and notify the GameLift service that the server is ready to host a game session. This code should run automatically before any GameLift-dependent code, such as on launch.

**Note**

Server API action names are the same in all available languages.

1. Initialize the Server SDK. Call `InitSdk()`.
2. Notify GameLift that a game server process is ready to host a game session. Each server process started on an GameLift instance must call `ProcessReady()` with the following information:
• Port number being used by the server process. This port number, along with an IP address, is stored in a game session object, which game clients use when connecting to an active game session.

• Location of files generated during an active game session that you want GameLift to retain, such as game session logs. Files generated by a server process when hosting a game session are saved to the GameLift instance, but lost when an instance is shut down. GameLift will upload any of these files you request; once uploaded, they are accessible through the GameLift console or by calling the GameLift API `GetGameSessionLogUrl()`. Consider using a file naming scheme that uniquely identifies game sessions if your fleet is configured to run multiple concurrent server processes per instance.

• Names of callback functions used by GameLift to trigger certain actions on the server process. More information on implementing these functions is provided in the following sections and in `ProcessParameters` (p. 245).
  
  • `onHealthCheck` (required) is called regularly to request a health status report from the server process.
  
  • `onStartGameSession` (required) is called when the GameLift service receives request to start a new game session (`CreateGameSession()`).
  
  • `onProcessTerminate` (required) is called when the GameLift service needs to force the server process to terminate, allowing the server process to shut down gracefully.
  
  • `onUpdateGameSession` (optional) is called when the GameLift service delivers an updated game session object to the game server or provides a status update on a match backfill request. This callback is required only if you're using the FlexMatch backfill (p. 15) feature. Learn more about how to implement this callback function at Update Match Data on the Game Server (p. 81).

Depending on how you configure your fleets, there may be multiple server processes running concurrently on a fleet instance. Once one server process on an instance calls `ProcessReady()` successfully, the GameLift service sets the instance’s status to ACTIVE.

You can set up a game server to securely access your resources on other AWS services. Learn more about possible ways to do this in Access AWS Resources From Your Fleets (p. 45).

Report Server Process Health

Add code to implement the callback function `onHealthCheck()`. This function is invoked by the GameLift service regularly to collect health metrics from the server process. The server process's response to a health check is a binary: healthy or unhealthy. When implementing this callback function, do the following:

• Evaluate the status of the server process using whatever measures make sense for your game. For example, you might report the server process as unhealthy if any external dependencies have failed or if metrics such as memory capacity fall outside a defined limit.

• Complete the health evaluation and respond to the callback within 60 seconds. If the GameLift service does not receive a response in that time, it will automatically consider the server process to be unhealthy.

• Return a boolean value: true for healthy, false for unhealthy.

If you do not implement a health check callback, the GameLift service considers the server process to be healthy unless the process is not responding, in which case it is considered unhealthy.

Server process health is used by GameLift to efficiently end unhealthy processes and free up resources. If a server process continues to report unhealthy or does not respond for three consecutive health checks,
the GameLift service may shut down the process and start a new one. Metrics on a fleet’s server process health is collected and viewable on the GameLift console.

**Start a Game Session**

Add code to implement the callback function `onStartGameSession`. This function is invoked by the GameLift service when creating a new game session, which may occur in response to a `CreateGameSession()` request or as part of game session placement or matchmaking activities.

The `onStartGameSession` function takes a GameSession object, provided by the GameLift service, as an input parameter. This object contains the game session ID and other information that defines the requested game session. The function should accomplish the following tasks:

- Perform whatever actions are needed to create a new game session. The new game session should reflect the GameSession object, including creating slots for the specified maximum number of players and referencing the game session name and ID. The GameLift service provides the same game session information to the game client.
- Process the game property values specified by the game client in its request. The game properties are contained in the GameSession object.
- At some point after the new game session is ready to accept players, the server process must call the server API action `ActivateGameSession()`. In response to a successful call, the GameLift service changes the game session status to ACTIVE.

**Retrieve a TLS Certificate**

If the game server is running on a fleet that has TLS certificate generation enabled, you can retrieve the TLS certificate and use it to establish a secure connection with a game client and encrypt client/server communication. A copy of the certificate is stored on the instance. To get the file location, call the section called “GetInstanceCertificate()” (p. 235).

**Validate a New Player**

Add code to verify a player connection request with the GameLift service. This code should run whenever a new player attempts to connect to the server process and before accepting the connection.

Connection requests from a game client must reference a player session ID. This ID is issued by the GameLift service and used to reserve a player slot in the game session (in response to an AWS SDK call to `CreatePlayerSession()`). The game server must call `AcceptPlayerSession()` with the referenced player session ID to verify that the connection request is coming from the authorized player.

Once the player session ID is validated by the GameLift service, the server process can accept the connection and allow the player to join the game session. If the player session ID is not validated by the GameLift service, the server process should deny the connection.

To acquire player data associated with the game session, call `DescribePlayerSessions()`.

**Report a Player Session Ending**

Add code to notify the GameLift service when a player disconnects from the game session. This code should run whenever the server process detects a dropped connection.

In the code handling the dropped connection, add a call to the server API action `RemovePlayerSession()` with the player session ID associated with the dropped connection. This notification enables the GameLift service to accurately track the number of current players and available slots in the game session.
Stop a Game Session

Add code to notify the GameLift service when a game session is ending. The notification enables the GameLift service to accurately track a server process's availability for new game sessions. This code should be added to the normal game session ending process.

At the end of the code to stop a game session, add a call to the server API action `TerminateGameSession()`. On successful receipt of this notification, the GameLift service changes the game session status to TERMINATED and may immediately start a new game session.

**Note**
If stopping a game session will be immediately followed by shutting down the server process, you can call the server API action `ProcessEnding()`, which terminates both the game session and the server process.

Shut Down a Server Process

Shut down may be initiated by a server process or by the GameLift service. Make the following changes to your game server code to handle either scenario.

- Implement the callback function `onProcessTerminate()`. This function should call your code that shuts down the game server. The GameLift service calls this function to initiate a server process shut down, usually prior to terminating the instance that is running the server process. Reasons to invoke this call is when GameLift is shutting down an unhealthy server process, when a spot instance is interrupted, or when instance capacity is scaling down. After receiving this call, the server process usually has a few minutes (two minutes in the case of a spot instance termination) to gracefully disconnect players, preserve game state data, and perform other cleanup tasks.

- Call the server API action `GetTerminationTime()` (p. 236) from your game server shut-down code. If GameLift has issued a call to terminate the server process, `GetTerminationTime()` returns the estimated termination time, if available. You can use this information to determine which shut-down activities can be done in the time remaining, such as saving game state data or migrating players to a new game session.

- At the start of your game server shut-down code, call the server API action `ProcessEnding()` (p. 237). This call notifies the GameLift service that the server process is shutting down. On receipt of this notification, the GameLift service changes the server process status to TERMINATED and may immediately recycle the instance's resources as needed. This also ensures that no new game sessions are sent to the process. Once `ProcessEnding()` has been invoked, it is safe for the process to shut down.

Access AWS Resources From Your Fleets

This topic describes how to set up your game servers and other applications to communicate directly and securely with other AWS resources while running on managed GameLift instances. Depending on your game structure, you may want applications or scripts in your game build to communicate with resources on other AWS services. For example, you might want to:

- Send instance log data to Amazon CloudWatch logs.
- Capture CloudWatch metrics to get better visibility into instance performance.
- Obtain sensitive information, such as passwords, that are stored remotely in an Amazon S3 account.
- Dynamically read and write game data that is stored in Amazon DynamoDB or other data storage service, such as game modes or inventory.
- Use Amazon SQS to send signals directly to an instance.
- Access custom resources that are deployed and running on Amazon EC2.
When you deploy game servers using GameLift, fleets and instances are allocated to your account but they are owned by the GameLift service. As a result, to enable access to AWS resources that are owned by your AWS account, you need to explicitly extend permissions to the Amazon GameLift service. In addition, you want to narrowly define the scope of the permissions being extended.

There are two potential methods available for securely accessing AWS resources from your hosted applications:

- Use an AWS Identity and Access Management (IAM) role to extend permissions to Amazon GameLift for your resources. This option is useful to access resources directly associated with AWS services, such as an Amazon S3 bucket, Amazon CloudWatch metrics, or AWS Lambdascripts. This is the simplest and recommended method.

- Use Amazon Virtual Private Cloud (VPC) peering connections to allow your Amazon GameLift fleets to communicate securely with AWS resources. This is an advanced feature and not a commonly used solution.

**Extend Access to GameLift Using Roles**

The following tasks must be completed to enable your game server or other applications to access your AWS resources while running on GameLift instances.

1. **Set up an IAM role for the GameLift service.** Follow the instructions at Set Up a Role for Amazon GameLift Access (p. 22) to create an IAM role. A role lays out a set of permissions for limited access to your AWS resources and specifies the entity (in this case the GameLift service) that can assume the role. Once you've created the role, take note of the new role's Amazon Resource Name (ARN), which you'll need when creating a fleet.

2. **Associate the service role with a GameLift fleet.** Once the service role is created, you need to enable your game servers and other applications to assume the service role while running on GameLift instances. To do this, you must provide the service role ARN to a fleet. Applications that are running on any instance in the fleet can then assume the role and acquire the necessary credentials for access.

   A service role can only be specified during fleet creation. Once the fleet is created, you cannot add or change the role ARN. Only one ARN can be provided to a fleet.

   For help with providing a service role ARN to a fleet, see Deploy a GameLift Fleet for a Custom Game Build (p. 104).

3. **Add code to your application to assume the service role.** Any application that is running on a GameLift instance can assume an IAM role, if one is associated with the fleet. This includes game servers and other executables, such as install scripts and daemons. An application can access resources during fleet creation and build installation, when starting or stopping service process and game sessions, or in response to game events.

   In the application code, before accessing an AWS resource, the application must first assume the service role by calling the AWS Security Token Service API AssumeRole and specifying the same service role ARN that is associated with the fleet. This action returns a set of temporary credentials, which enable the application to access the AWS resource. Learn more about Using Temporary Security Credentials to Request Access to AWS Resources.

**Use VPC Peering with Amazon GameLift Fleets**

You can use Amazon Virtual Private Cloud (VPC) peering to establish fast and secure communication between an application running on a GameLift instance and another AWS resource. An Amazon VPC is a virtual network, defined by you, that includes a set of resources managed through your AWS account. Each GameLift fleet has its own VPC. With VPC peering, you can establish a direct network connection between the VPC for your fleet and the VPC for your other AWS resources.
For example, you might have a set of web services that support your game, such as for player authentication or social networking. You can set up a VPC for these resources, and then use GameLift’s VPC peering to enable your game servers to make direct network calls to the web services. With VPC peering, calls from your game server processes incur minimal latency and, since they are not routed over the public Internet, are not exposed externally.

GameLift streamlines the process of setting up VPC peering connections for your game servers. It handles peering requests, updates route tables, and configures the connections as required. For more information on how to set up VPC peering for your game servers, see VPC Peering for Amazon GameLift (p. 165).

See more information on Amazon’s virtual private clouds and VPC peering. You can peer your GameLift fleets with VPCs in any AWS account that you have access to.

Integrating your Game Client for Amazon GameLift

The topics in this section describe the managed GameLift functionality that you can add to a game client or game service. A game client or service may need to handle the following tasks:

- Request information about active game sessions from the GameLift service.
- Join a player to an existing game session.
- Create a new game session and join players to it.
- Change metadata about an existing game session.

Adding GameLift to your multiplayer game client is Step 5 in the Get Started with Custom Servers (p. 30). The following instructions assume that you've created an AWS account, generated an GameLift-enabled game server and uploaded it to GameLift, and used GameLift tools (such as the GameLift console) to create and configure a virtual fleet to host your game sessions. When adding GameLift to your game client, you must be able to provide AWS account credentials and specify a fleet to be used with the client.

For more information on how game clients interact with the GameLift service and game servers running on GameLift, see Amazon GameLift and Game Client/Server Interactions (p. 51).

Topics

- Add Amazon GameLift to Your Game Client (p. 47)
- Create Game Sessions with Queues (p. 51)
- Generate Player IDs (p. 51)

Add Amazon GameLift to Your Game Client

When using the managed GameLift solution, you can integrate GameLift into game components that need to acquire game session information, create new game sessions, and/or join players to games. Depending on your game architecture, this functionality might be placed in the game client or in game services that handle tasks such as player authentication, matchmaking, or game placement.

To do this, use the AWS SDK with GameLift API. This SDK is available in C++, C#, and several other languages. For details on the AWS SDK, version information, and language support, see For Client Services (p. 24). Most of the links in this topic go to the GameLift Service API Reference, which describes the low-level service API for GameLift-related actions and includes links to language-specific reference guides.

Note

Interested in using matchmaking to get players into games? Once you've set up GameLift on your game client or a service, add FlexMatch to match players and create custom game sessions. See more at Adding FlexMatch Matchmaking (p. 69).
Set Up GameLift on a Client or Service

You add code to initialize an GameLift client and store some key settings for use with GameLift. This code needs to be located so that it runs before any GameLift-dependent code, such as on launch.

Note
To set up your game client for testing using GameLift Local, see Testing Your Integration (p. 54)

1. Decide whether to use the default client configuration or create custom settings. For custom settings, you must create a custom ClientConfiguration object. See AWS ClientConfiguration (C++) for object structure and the default settings.

A client configuration specifies a target region and endpoint. The region determines which resources (fleets, queues, matchmakers, etc.) GameLift interacts with when responding to requests. The default client configuration specifies the US East (N. Virginia) region. To use any other region, create a custom configuration. See this list of AWS regions supported by GameLift for names and endpoints. If your client or service needs to make requests for multiple regions, create a separate client configuration object for each target region and each as needed. See Using Regions with the AWS SDKs for language-specific examples.

2. Initialize an GameLift client. Call Aws::GameLift::GameLiftClient() (C++) using either a client configuration with the default settings or a custom configuration.

3. Add a mechanism to generate a unique identifier for each player. GameLift requires a unique player ID to connect to a game session. For more details, see Generate Player IDs (p. 51).

4. Collect and store the following information to use when contacting GameLift:

- **Target fleet** – Most GameLift API requests must specify a fleet, such as when getting information on available game sessions or managing game sessions and player sessions. How you define the optimal target fleet (for example, setting a static fleet, or choosing a fleets based on a device’s physical location). To specify a target fleet, use either a fleet ID or an alias ID that points to the target fleet. Fleet aliases are highly useful in that you can switch players from one fleet to another without issuing a game client update. The combination of target fleet and region (specified in the client configuration) uniquely identifies the fleet.

- **Target queue** – If your game uses multi-fleet queues to place new game sessions, you can specify which queue to use. To specify a target queue, use the queue name. The queue must be configured in the region

- **AWS credentials** – All calls to the GameLift service must provide credentials for the AWS account that hosts the game. This is the account you used to set up your GameLift fleets, and you should have created an IAM user or user group for players with a permissions policy. You need to create an Aws::Auth::AWSCredentials (C++) object containing an IAM access key and secret key for the player user group. For help finding the keys, see Managing Access Keys for IAM Users.

Get Game Sessions

Add code to discover available game sessions and manage game sessions settings and metadata. See Game and Player Session Features (p. 12) for more on game session features.

Search for active game sessions.

Use SearchGameSessions to get information on a specific game session, all active sessions, or sessions that meet a set of search criteria. This call returns a GameSession object for each active game session that matches your search request.

Use search criteria to get a filtered list of active game sessions for players to join. For example, you can filter sessions as follows:
• Exclude game sessions that are not accepting new players:
  PlayerSessionCreationPolicy = DENY_ALL
• Exclude game sessions that are full:
  CurrentPlayerSessionCount = MaximumPlayerSessionCount
• Choose game sessions based on length of time the session has been running:
  Evaluate CreationTime
• Find game sessions based on a custom game property:
  gameSessionProperties.gameMode = "brawl"

Manage game sessions.

Use any of the following operations to retrieve or update game session information.

- DescribeGameSessionDetails() – Get a game session's protection status in addition to game session information.
- UpdateGameSession() – Change a game session's metadata and settings as needed.
- GetGameSessionLogUrl – Access stored game session logs.

Create Game Sessions

Add code to start new game sessions on your deployed fleets and make them available to players. There are two options for creating game sessions, depending on whether you are deploying your game in multiple regions or in a single region.

Create a game session using a multi-region queue.

Use StartGameSessionPlacement to place a request for a new game session in a queue. To use this feature, you'll need to set up a queue, which determines where (on which fleets) the new game session can be placed. A queue processes a game session placement request by trying each possible fleet, in turn, until it finds one with available resources to host the new game session. For more detailed information on queues and how to use them, see Design a Game Session Queue (p. 128).

When creating a game session placement, specify the name of the queue to use, a game session name, a maximum number of concurrent players for the game, and an optional set of game properties. You can optionally provide a list of players to automatically join to the game session. If you include player latency data for relevant regions, GameLift uses this information to place the new game session on a fleet that provides the best possible gameplay experience for players.

Game session placement is an asynchronous process. Once you've placed a request, you can let it succeed or time out. You can also cancel the request at any time using StopGameSessionPlacement. To check the status of your placement request, call DescribeGameSessionPlacement to retrieve an updated GameSessionPlacement object. When a new game session is created, the GameSessionPlacement reflects the following changes: (1) Status changes from Pending to Fulfilled; (2) New game session information is added, including game session ID and region; and (3) New player session information is added (if requested).

Create a game session on a specific fleet.

Use CreateGameSession to create a new session on a specified fleet. This synchronous operation succeeds or fails depending on whether the fleet has resources available to host a new game session. Your game should handle failures as best suits your game and players. For example, you might repeat the request until resources are freed or scaled up, or you might switch to a different fleet. Once GameLift has created the new game session and returned a GameSession object, you can start joining players to it.

When you use this method to create a game session, specify a fleet ID or alias ID, a session name, and a maximum number of concurrent players for the game. Optionally, you can include a set of game properties. Game properties are defined in an array of key–value pairs in which you define the keys and a
set of values that are meaningful to your game. This information is passed to the server process hosting the new game session, to be used as designed in your game server. For example, you might use game properties to direct the game session to use a certain game map or a particular set of rules.

If you use the GameLift resource protection feature to limit the number of game sessions one player can create, you'll need to specify the game session creator's player ID.

**Join a Player to a Game Session**

Add code to reserve player slots in active game sessions and connect game clients to game sessions.

1. **Reserve a player slot in a game session.**

   To reserve a player slot, create a new player session for the game session. See How Players Connect to Games (p. 12) for more on player sessions. You have two ways to create new player sessions:
   
   - If you're using `StartGameSessionPlacement` to create game sessions, as described in the previous section, you can reserve slots for one or more players in the new game session.
   - Reserve player slots for one or more players using `CreatePlayerSession` or `CreatePlayerSessions` with a game session ID.

   With both methods, GameLift first verifies that the game session is accepting new players and has an available player slot. If successful, GameLift reserves a slot for the player, creates the new player session, and returns a `PlayerSession` object containing the DNS name, IP address, and port that a game client needs to connect to the game session.

   A player session request must include a unique ID for each player. See Generate Player IDs (p. 51) for more on player IDs.

   Optionally, a player session request can include a set of custom player data. This data is stored in the newly created player session object, which can be retrieved by calling `DescribePlayerSessions()`. It is also passed from the GameLift service to the game server when the player connects directly to the game session. Player data is not used by GameLift; it is a simple string of characters that is available to your game components for interpretation. When requesting multiple player sessions, you can provide a string of player data for each player, mapped to the player ID in the request.

2. **Connect to a game session.**

   Add code to the game client to retrieve the `PlayerSession` object, which contains the game session's connection information. Use this information to establish a direct connection to the server process.

   - You can connect using the specified port and either the DNS name or IP address assigned to the server process.

     **Note**
     
     If your fleets have TLS certificate generation enabled, you must connect using the DNS name and port. This is required even if you haven't implemented a server authentication process.

   - If you're using player session IDs to reserve player slots and track player connections, you must reference the player session ID.

   Once connected, the game client and server process communicate directly without involving the GameLift service to communicate gameplay. The server process maintains communication with the GameLift service to report player connection status, health status, etc. On initial connection, it may contact the service to verify that the player session ID is valid and in a reserved status. If validated, the reservation is claimed and the player connection is accepted. Later, when the player disconnects, the server process reports the dropped connection.
If you want to enable server authentication and encrypt data packets travelling between game client and the game session, you need to build this functionality. When the TLS certificate generation feature is enabled for a new fleet, GameLift only gets the TLS certificate and creates DNS entries for each instance in the fleet.

Create Game Sessions with Queues

This set of features lets you place new game sessions more efficiently across Amazon GameLift resources, and better supports matchmaking services. Previously, new game session requests were limited to a single fleet (CreateGameSession), and the request failed if the fleet was at full capacity or otherwise compromised.

Use a queue to place new game sessions on any one of a group of fleets that can span regions. Increased player demand can be shifted to lesser used fleets in other regions as needed. Queues also decrease the overhead needed to monitor fleets and balance resources across multiple fleets and regions. You can manage queues and track queue performance metrics in the Amazon GameLift Console.

Create a new game session placement request and add it to a queue. A game session placement request includes standard game session properties, and also lets you add one or more players to the new game session.

When creating a game session placement request, include player latency data to help Amazon GameLift choose a fleet in a region that provides the best possible experience for all the players.

Generate Player IDs

The managed GameLift service uses a player session to represent a player connected to a game session. A player session must be created each time a player connects to a game session. When a player leaves a game, the player session ends and is not reused.

GameLift provides a file called Lobby.cpp in the Amazon Lumberyard sample project MultiplayerSample that demonstrates how to generate a new, random ID number for every player in every new game session. You are not required to use the sample code; we provide it as an example. You can also rewrite the code to persist your own unique, non-personally identifiable player IDs.

The following sample code in Lobby.cpp randomly generates unique player IDs:

```cpp
bool includeBrackets = false;
bool includeDashes = true;
string playerId = AZ::Uuid::CreateRandom().ToString<string>(includeBrackets, includeDashes);
```

You can view player sessions by Player ID in the AWS Management Console for GameLift. For more information on player sessions, see View Data on Game and Player Sessions (p. 176).

Amazon GameLift and Game Client/Server Interactions

This topic describes the interactions between a client app, a game server, and the managed GameLift service. See also the Amazon GameLift–Game Server/Client Interactions (p. 53) diagram.

Setting Up a New Server Process

1. The GameLift service launches a new server process on an Amazon Elastic Compute Cloud (Amazon EC2) instance.
2. The server process, as part of the launch process, calls these server API actions:
   - `InitSDK()` to initialize the server SDK.
   - `ProcessReady()` to communicate readiness to accept a game session and specify connection port and location of game session log files.

   It then waits for a callback from the GameLift service.

3. The GameLift service changes the status of the EC2 instance to ACTIVE, with 0 game sessions and 0 players.

4. The GameLift service begins calling the `onHealthCheck` callback regularly while the server process is active. The server process must report either healthy or not healthy within one minute.

## Creating a Game Session

1. The Client app calls the client API action `CreateGameSession()`.

2. The GameLift service searches for an active server with 0 game sessions. When found, it does the following:
   - Creates a new `GameSession` object, using the port setting reported by the server process in `ProcessReady()`, and sets its status to ACTIVATING.
   - Responds to the client app request with the `GameSession` object.
   - Invokes the `onStartGameSession` callback on the server process, passing the `GameSession` object.

3. The server process runs the `onStartGameSession` callback function. When ready to accept player connections, the server process calls the server API action `ActivateGameSession()` and waits for player connections.

4. The GameLift service changes the `GameSession` status to ACTIVE.

## Adding a Player to a Game Session

1. The Client app calls the client API action `CreatePlayerSession()` with a game session ID.

2. The GameLift service checks the game session status (must be ACTIVE), and looks for an open player slot in the game session. If a slot is available, it does the following:
   - Creates a new `PlayerSession` object and sets its status to RESERVED.
   - Responds to the client app request with the `PlayerSession` object.

3. The Client app connects directly to the server process with the player session ID.

4. The server process calls the Server API action `AcceptPlayerSession()` to validate the player session ID. If validated, the GameLift service passes the `PlayerSession` object to the server process. The server process either accepts or rejects the connection.

5. The GameLift service does one of the following:
   - If the connection is accepted, sets the `PlayerSession` status to ACTIVE.
   - If no response is received within 60 seconds of the client app's original `CreatePlayerSession()` call, changes the `PlayerSession` status to TIMEDOUT and reopens the player slot in the game session.

## Removing a Player From a Game Session

1. The Client app disconnects from the server process.

2. The server process detects the lost connection and calls the server API action `RemovePlayerSession()`.
3. The GameLift service changes the PlayerSession status to COMPLETED and reopens the player slot in the game session.

Shutting down a Game Session

Shutting down a game session
1. The server process calls the server API action TerminateGameSession().
2. The GameLift service does the following:
   • Changes the GameSession status to TERMINATED.
   • Uploads game session logs to Amazon Simple Storage Service (Amazon S3).
   • Updates fleet utilization to indicate the server is idle (0 game sessions, 0 players).

Terminating a Server Process

1. The server process does the following:
   • Runs code that gracefully shuts down the server process.
   • Calls the server API action ProcessEnding() to inform the GameLift service.
2. The GameLift service does the following:
   • Uploads game session logs (if any) to Amazon S3.
   • Changes the server process status to TERMINATED.
   • Recycles instance resources based on the fleet’s runtime configuration.

Responding to a Shutdown Request

1. The GameLift service invokes the server process’s onProcessTerminate callback. This call is used to shut down a server process that has reported unhealthy or not responded with health status for three consecutive minutes.
2. The server process runs the onProcessTerminate callback function, which triggers the server’s termination process, ending with a call to ProcessEnding().
3. The GameLift service does the following, either in response to receiving the ProcessEnding() call or after five minutes:
   • Uploads game session logs (if any) to Amazon S3.
   • Changes the server process status to TERMINATED.
   • Recycles instance resources based on the fleet’s runtime configuration.

Amazon GameLift–Game Server/Client Interactions

The diagram below outlines API interactions that occur when managing multiplayer game session tasks using the managed GameLift solution.
Testing Your Integration

Use GameLift Local to run a limited version of the managed GameLift service on a local device and test your game integration against it. This tool is useful when doing iterative development on your game integration. The alternative—uploading each new build to GameLift and configuring a fleet to host your game—can take 30 minutes or more each time.

With GameLift Local, you can verify the following:

- Your game server is correctly integrated with the Server SDK and is properly communicating with the GameLift service to start new game sessions, accept new players, and report health and status.
• Your game client is correctly integrated with the AWS SDK for GameLift and is able to retrieve information on existing game sessions, start new game sessions, join players to games and connect to the game session.

GameLift Local is a command-line tool that starts a self-contained version of the managed GameLift service. GameLift Local also provides a running event log of server process initialization, health checks, and API calls and responses. GameLift Local recognizes a subset of the AWS SDK actions for GameLift. You can make calls from the AWS CLI or from your game client. All API actions perform locally just as they do in the GameLift web service.

**Set Up GameLift Local**

GameLift Local is provided as an executable .jar file bundled with the Server SDK. It can be run on Windows or Linux and used with any GameLift-supported language.

Before running Local, you must also have the following installed.

• A build of the GameLift Server SDK version 3.1.5 or higher
• Java 8

**Test a Game Server**

If you want to test your game server only, you can use the AWS CLI to simulate game client calls to the GameLift Local service. This verifies that your game server is performing as expected with the following:

• The game server launches properly and initializes the GameLift Server SDK.
• As part of the launch process, the game server notifies GameLift that the server is ready to host game sessions.
• The game server sends health status to GameLift every minute while running.
• The game server responds to requests to start a new game session.

1. **Start GameLift Local.**

   Open a command prompt window, navigate to the directory containing the file GameLiftLocal.jar and run it. By default, Local listens for requests from game clients on port 8080. To specify a different port number, use the -p parameter, as shown in the following example:

   ```java -jar GameLiftLocal.jar -p 9080```

   Once Local starts, you see logs indicating that two local servers were started, one listening for your game server and one listening for your game client or the AWS CLI. Logs continue to report activity on the two local servers, including communication to and from your game components.

2. **Start your game server.**

   Start your GameLift-integrated game server locally. You don't need to change the endpoint for the game server.

   In the Local command prompt window, log messages indicate that your game server has connected to the GameLift Local service. This means that your game server successfully initialized the GameLift Server SDK (with InitSDK()). It has called ProcessReady() with the log paths shown and, if successful, is ready to host a game session. While the game server is running, GameLift logs each health status report from the game server. The following log messaging example shows a successfully integrated game server:
Potential error and warning messages include the following:

- Error: “ProcessReady did not find a process with pID: <process ID>! Was InitSDK() invoked?”
- Warning: “Process state already exists for process with pID: <process ID>! Is ProcessReady(...) invoked more than once?”

3. **Start the AWS CLI.**

Once your game server successfully calls `ProcessReady()`, you can start making client calls. Open another command prompt window and start the AWS CLI tool. Get and install the AWS Command Line Interface tool. The AWS CLI by default uses the GameLift web service endpoint. You must override this with the Local endpoint in every request using the `--endpoint-url` parameter, as shown in the following example request.

```
aws gamelift describe-game-sessions --endpoint-url http://localhost:9080 --fleet-id fleet-123
```

In the AWS CLI command prompt window, `aws gamelift` commands result in responses as documented in the AWS CLI Command Reference.

4. **Create a game session.**

With the AWS CLI, submit a `CreateGameSession()` request. The request should follow the expected syntax. For Local, the `FleetId` parameter can be set to any valid string (`^fleet-\S+$`).

```
aws gamelift create-game-session --endpoint-url http://localhost:9080 --maximum-player-session-count 2 --fleet-id fleet-la2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d
```

In the Local command prompt window, log messages indicate that GameLift Local has sent your game server an `onStartGameSession` callback. If a game session is successfully created, your game server responds by invoking `ActivateGameSession`.

```
13:57:36,129  INFO || - [SDKInvokerImpl] Thread-2 - Finished sending event to game server to start a game session:
arn:aws:gamelift:local:gamesession/fleet-la2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d/gamesess-ab423a4b-b827-4765-aae2-54b3f0a81b6.
Waiting for ack response. 13:57:36,143  INFO || - [SDKInvokerImpl] Thread-2 - Received ack response: true13:57:36,144  INFO || - [CreateGameSessionDispatcher] Thread-2 - GameSession with id:
```

```
Version
56
```
In the AWS CLI window, GameLift responds with a game session object including a game session ID. Notice that the new game session's status is Activating. The status changes to Active once your game server invokes ActivateGameSession. If you want to see the changed status, use the AWS CLI to call DescribeGameSessions().

```json
{
    "GameSession": {
        "Status": "ACTIVATING",
        "MaximumPlayerSessionCount": 2,
        "FleetId": "fleet-1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d",
        "GameSessionId": "arn:aws:gamelift:local::gamesession/fleet-1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d/gsess-abcdef12-3456-7890-abcd-ef1234567890",
        "IpAddress": "127.0.0.1",
        "Port": 1935
    }
}
```

**Test a Game Server and Client**

To check your full game integration, including connecting players to games, you can run both your game server and client locally. This allows you to test programmatic calls from your game client to the GameLift Local. You can verify the following actions:

- The game client is successfully making AWS SDK requests to the GameLift Local service, including to create game sessions, retrieve information on existing game sessions, and create player sessions.
- The game server is correctly validating players when they try to join a game session. For validated players, the game server may retrieve player data (if implemented).
- The game server reports a dropped connection when a player leaves the game.
- The game server reports ending a game session.

1. **Start GameLift Local.**

   Open a command prompt window, navigate to the directory containing the file `GameLiftLocal.jar` and run it. By default, Local listens for requests from game clients on port 8080. To specify a different port number, use the `--p` parameter, as shown in the following example.

   ```shell
   ./gamelft-local -p 9080
   ```

   Once Local starts, you see logs showing that two local servers were started, one listening for your game server and one listening for your game client or the AWS CLI.

2. **Start your game server.**

   Start your GameLift-integrated game server locally. See **Test a Game Server (p. 55)** for more detail on message logs.

3. **Configure your game client for Local and start it.**
To use your game client with the GameLift Local service, you must make the following changes to your game client’s setup, as described in Set Up GameLift on a Client or Service (p. 48):

- Change the `ClientConfiguration` object to point to your Local endpoint, such as `http://localhost:9080`.
- Set a target fleet ID value. For Local, you do not need a real fleet ID; set the target fleet to any valid string (`^fleet-\S+$`), such as `fleet-1a2b3c4d-5e6f-7a8b-9c0d-1e2f3a4b5c6d`.
- Set AWS credentials. For Local, you do not need real AWS credentials; you can set the access key and secret key to any string.

In the Local command prompt window, once you start the game client, log messages should indicate that it has initialized the `GameLiftClient` and is successfully communicated with the GameLift service.

4. **Test game client calls to the GameLift service.**

Verify that your game client is successfully making any or all of the following API calls:

- `CreateGameSession()`
- `DescribeGameSessions()`
- `CreatePlayerSession()`
- `CreatePlayerSessions()`
- `DescribePlayerSessions()`

In the Local command prompt window, only calls to `CreateGameSession()` result in log messages. Log messages show when GameLift Local prompts your game server to start a game session (`onStartGameSession` callback) and gets a successful `ActivateGameSession` when your game server invokes it. In the AWS CLI window, all API calls result in responses or error messages as documented.

5. **Verify that your game server is validating new player connections.**

After creating a game session and a player session, establish a direct connection to the game session.

In the Local command prompt window, log messages should show that the game server has sent an `AcceptPlayerSession()` request to validate the new player connection. If you use the AWS CLI to call `DescribePlayerSessions()`, the player session status should change from Reserved to Active.

6. **Verify that your game server is reporting game and player status to the GameLift service.**

For GameLift to manage player demand and correctly report metrics, your game server must report various statuses back to GameLift. Verify that Local is logging events related to following actions. You may also want to use the AWS CLI to track status changes.

- **Player disconnects from a game session** – GameLift Local log messages should show that your game server calls `RemovePlayerSession()`. An AWS CLI call to `DescribePlayerSessions()` should reflect a status change from Active to Completed. You might also call `DescribeGameSessions()` to check that the game session’s current player count decreases by one.

- **Game session ends** – GameLift Local log messages should show that your game server calls `TerminateGameSession()`. An AWS CLI call to `DescribeGameSessions()` should reflect a status change from Active to Terminated (or Terminating).

- **Server process is terminated** – GameLift Local log messages should show that your game server calls `ProcessEnding()`.

---

Version

58
Variations with Local

When using GameLift Local, keep in mind the following:

- Unlike the GameLift web service, Local does not track a server's health status and initiate the onProcessTerminate callback. Local simply stops logging health reports for the game server.
- For calls to the AWS SDK, fleet IDs are not validated, and can be any string value that meets the parameter requirements (`^fleet-\S+$`).
- Game session IDs created with Local have a different structure. They include the string `local`, as shown here:

  arn:aws:gamelift:local::gamesession/fleet-123/gsess-56961f8e-db9c-4173-97e7-270b82f0daa6

Integrating Games with Amazon GameLift Realtime Servers

Amazon GameLift Realtime Servers offers a lightweight server solution that is designed for games that don't need a complex, custom-built game server. Games such as mobile games, turn-based games, message-based games, etc., can use ready-to-go Realtime game servers that require minimal configuration but can also be customized with game-specific logic.

Tip
Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using sample games (p. 29).

The topics in this section describe how to enable your multiplayer game clients to connect to and use Realtime Servers game servers that are running on GameLift. For a complete roadmap to getting your game up and running with Realtime Servers, see Get Started with Realtime Servers (p. 31).

Learn more about Realtime Servers:

- How Realtime Servers Work (p. 8)

Get started integrating your game with Realtime Servers:

- Integrating a Game Client for Realtime Servers (p. 59)
- Creating a Realtime Script (p. 63)

- Get Started with Realtime Servers (p. 31)

Integrating a Game Client for Realtime Servers

This topic describes how to prepare your game client to be able to join and participate in Amazon GameLift-hosted game sessions.

There are two sets of tasks needed to prepare your game client:

- Set up your game client to acquire information about existing games, request matchmaking, start new game sessions, and reserve game session slots for a player.
- Enable your game client to join a game session hosted on a Realtime server and exchange messages.
Find or Create Game Sessions and Player Sessions

Set up your game client to find or start game sessions, request FlexMatch matchmaking, and reserve space for players in a game by creating player sessions. As a best practice, create a client service and use it to make the direct requests to the Amazon GameLift service when triggered by a game client action. The client service then relays relevant responses back to the game client.

1. Add the AWS SDK to your client service project, initialize an Amazon GameLift client, and configure it to use the hosting resources in your fleets and/or queues. The AWS SDK is available in several languages; see the Amazon GameLift SDKs For Client Services (p. 24).

2. Add GameLift functionality to your client service. For more detailed instructions, see Add Amazon GameLift to Your Game Client (p. 47) and Adding FlexMatch Matchmaking (p. 69). The best practice is to use game session placements to create new game sessions. This method lets you take full advantage of GameLift's ability to quickly and intelligently place new game sessions, as well as use player latency data to minimize game lag. At a minimum, your client service must be able to request new game sessions and handle game session data in response. You may also want to add functionality to search for and get information on existing game sessions, and request player sessions, which effectively reserve a player slot in an existing game session.

3. Convey connection information back to the game client. The backend game service receives game session and player session objects in response to requests to the Amazon GameLift service. These objects contain information, in particular connection details (IP address and port) and player session ID, that the game client needs to connect to the game session running on a Realtime Server.

Connect to Games on Realtime Servers

Enable your game client to connect directly with a hosted game session on a Realtime server and exchange messages with the server and with other players.

1. Get the Realtime Client SDK, build it, and add it to your game client project. See the README file for more information on SDK requirements and instructions on how to build the client libraries.

2. Call Client() (p. 218) with a client configuration that specifies the type of client/server connection to use.

   **Note**
   If you’re connecting to a Realtime server that is running on a secured fleet with a TLS certificate, you must specify a secured connection type.

3. Add the following functionality to your game client. See the Realtime Servers Client API (C#) Reference (p. 217) for more information.
   - Connect to and disconnect from a game
     - Connect() (p. 218)
     - Disconnect() (p. 219)
   - Send messages to target recipients
     - SendMessage() (p. 220)
   - Receive and process messages
     - OnDataReceived() (p. 223)
   - Join groups and leave player groups
     - JoinGroup() (p. 220)
     - RequestGroupMembership() (p. 221)
     - LeaveGroup() (p. 220)

4. Set up event handlers for the client callbacks as needed. See Realtime Servers Client API (C#) Reference: Asynchronous Callbacks (p. 221).
When working with Realtime fleets that have TLS certificate generation enabled, the server is automatically authenticated using the TLS certificate. TCP and UDP traffic is encrypted in flight to provide transport layer security. TCP traffic is encrypted using TLS 1.2, and UDP traffic is encrypted using DTLS 1.2.

**Game Client Examples**

**Basic Realtime Client (C#)**

This example illustrates a basic game client integration with the Realtime Client SDK (C#). As shown, the example initializes a Realtime client object, sets up event handlers and implements the client-side callbacks, connects to a Realtime server, sends a message, and disconnects.

```csharp
using System;
using System.Text;
using Aws.GameLift.Realtime;
using Aws.GameLift.Realtime.Event;
using Aws.GameLift.Realtime.Types;

namespace Example
{
    class RealTimeClient
    {
        public Aws.GameLift.Realtime.Client Client { get; private set; }

        // An opcode defined by client and your server script that represents a custom message type
        private const int MY_TEST_OP_CODE = 10;

        /// Initialize a client for GameLift Realtime and connect to a player session.
        /// <param name="endpoint">The DNS name that is assigned to Realtime server</param>
        /// <param name="remoteTcpPort">A TCP port for the Realtime server</param>
        /// <param name="listeningUdpPort">A local port for listening to UDP traffic</param>
        /// <param name="connectionType">Type of connection to establish between client and the Realtime server</param>
        /// <param name="playerSessionId">The player session ID that is assigned to the game client for a game session</param>
        /// <param name="connectionPayload">Developer-defined data to be used during client connection, such as for player authentication</param>
        public RealTimeClient(string endpoint, int remoteTcpPort, int listeningUdpPort, ConnectionType connectionType, string playerSessionId, byte[] connectionPayload)
        {
            // Create a client configuration to specify a secure or unsecure connection type
            ClientConfiguration clientConfiguration = new ClientConfiguration()
            {
                // Best practice is to set up a secure connection using the connection type RT_OVER_WSS_DTLS_TLS12.
                ConnectionType = connectionType
            };

            // Create a Realtime client with the client configuration
```

Version
61
Client = new Client(clientConfiguration);

// Initialize event handlers for the Realtime client
Client.ConnectionOpen += OnOpenEvent;
Client.ConnectionClose += OnCloseEvent;
Client.GroupMembershipUpdated += OnGroupMembershipUpdate;
Client.DataReceived += OnDataReceived;

// Create a connection token to authenticate the client with the Realtime server
// Player session IDs can be retrieved using AWS SDK for GameLift
ConnectionToken connectionToken = new ConnectionToken(playerSessionId, connectionPayload);

// Initiate a connection with the Realtime server with the given connection information
Client.Connect(endpoint, remoteTcpPort, listeningUdpPort, connectionToken);

public void Disconnect()
{
    if (Client.Connected)
    {
        Client.Disconnect();
    }
}

public bool IsConnected()
{
    return Client.Connected;
}

/// <summary>
/// Example of sending to a custom message to the server.
/// Server could be replaced by known peer Id etc.
/// </summary>
/// <param name="intent">Choice of delivery intent ie Reliable, Fast etc.</param>
/// <param name="payload">Custom payload to send with message</param>
public void SendMessage(DeliveryIntent intent, string payload)
{
    Client.SendMessage(Client.NewMessage(MY_TEST_OP_CODE)
        .WithDeliveryIntent(intent)
        .WithTargetPlayer(Constants.PLAYER_ID_SERVER)
        .WithPayload(StringToBytes(payload)));
}

/**
 * Handle connection open events
 */
public void OnOpenEvent(object sender, EventArgs e)
{
}

/**
 * Handle connection close events
 */
public void OnCloseEvent(object sender, EventArgs e)
{
}

/**
 * Handle Group membership update events
 */
public void OnGroupMembershipUpdate(object sender, GroupMembershipEventArgs e)
{
Creating a Realtime Script

To use Realtime Servers for your game, you need to provide a script (in the form of some JavaScript code) to configure and optionally customize a fleet of Realtime Servers. This topic covers the key steps in creating a Realtime script. Once the script is ready, upload it to the Amazon GameLift service and use it to create a fleet (see Upload a Realtime Servers Script to Amazon GameLift (p. 96)).

To prepare a script for use with Realtime Servers, add the following functionality to your Realtime script.

**Manage Game Session Life-Cycle (required)**

At a minimum, a Realtime script must include the `Init()` function, which prepares the Realtime server to start a game session. It is also highly recommended that you also provide a way to terminate game sessions, to ensure that new game sessions can continue to be started on your fleet.

The `Init()` callback function, when called, is passed a Realtime session object, which contains an interface for the Realtime server. See Realtime Servers Interface (p. 229) for more details on this interface.

To gracefully end a game session, the script must also call the Realtime server’s `session.processEnding` function. This requires some mechanism to determine when to end a session. The script example code illustrates a simple mechanism that checks for player connections and triggers game session termination when no players have been connected to the session for a specified length of time.

Realtime Servers with the most basic configuration—server process initialization and termination—essentially act as stateless relay servers. The Realtime server relays messages and game data between
game clients that are connected to the game, but takes no independent action to process data or perform logic. You can optionally add game logic, triggered by game events or other mechanisms, as needed for your game.

### Add Server-Side Game Logic (optional)

You can optionally add game logic to your Realtime script. For example, you might do any or all of the following. The script example code provides illustration. See Amazon GameLift Realtime Servers Script Reference (p. 227).

- **Add event-driven logic.** Implement the callback functions to respond to client-server events. See

  You can provide custom logic to respond to events by implementing these callbacks in your Realtime script.

  **init**

  Initializes the Realtime server and receives a Realtime server interface.

  **Syntax**

  ```
  init(rtsession)
  ```

  **onMessage**

  Invoked when a received message is sent to the server.

  **Syntax**

  ```
  onMessage(gameMessage)
  ```

  **onHealthCheck**

  Invoked to set the status of the game session health. By default, health status is healthy (or `true`). This callback can be implemented to perform custom health checks and return a status.

  **Syntax**

  ```
  onHealthCheck()
  ```

  **onStartGameSession**

  Invoked when a new game session starts, with a game session object passed in.

  **Syntax**

  ```
  onStartGameSession(session)
  ```

  **onProcessTerminate**

  Invoked when the server process is being terminated by the Amazon GameLift service. This can act as a trigger to exit cleanly from the game session. There is no need to call `processEnding()`.
Syntax

```plaintext
onProcessTerminate()
```

**onPlayerConnect**

Invoked when a player requests a connection and has passed initial validation.

Syntax

```plaintext
onPlayerConnect(connectMessage)
```

**onPlayerAccepted**

Invoked when a player connection is accepted.

Syntax

```plaintext
onPlayerAccepted(player)
```

**onPlayerDisconnect**

Invoked when a player disconnects from the game session, either by sending a disconnect request or by other means.

Syntax

```plaintext
onPlayerDisconnect(peerId)
```

**onProcessStarted**

Invoked when a server process is started. This callback allows the script to perform any custom tasks needed to prepare to host a game session.

Syntax

```plaintext
onProcessStarted(args)
```

**onSendToPlayer**

Invoked when a message is received on the server from one player to be delivered to another player. This process runs before the message is delivered.

Syntax

```plaintext
onSendToPlayer(gameMessage)
```

**onSendToGroup**

Invoked when a message is received on the server from one player to be delivered to a group. This process runs before the message is delivered.
Customizing a Realtime Script

### Syntax

```javascript
onSendToGroup(gameMessage))
```

### onPlayerJoinGroup

**Invoked when a player sends a request to join a group.**

**Syntax**

```javascript
onPlayerJoinGroup(groupId, peerId)
```

### onPlayerLeaveGroup

**Invoked when a player sends a request to leave a group.**

**Syntax**

```javascript
onPlayerLeaveGroup(groupId, peerId)
```

(p. 227) for a complete list of callbacks.

- **Trigger logic by sending messages to the server.** Create a set of special operation codes for messages sent from game clients to the server, and add functions to handle receipt. Use the callback `onMessage`, and parse the message content using the `gameMessage` interface (see `gameMessage.opcode (p. 232)`).

### Realtime Servers Script Example

This example illustrates a basic script needed to deploy Realtime Servers plus some custom logic. It contains the required `Init()` function, and uses a timer mechanism to trigger game session termination based on length of time with no player connections. It also includes some hooks for custom logic, including some callback implementations.

```javascript
// Example Realtime Server Script
'use strict';

// Example override configuration
const configuration = {
  pingIntervalTime: 30000
};

// Timing mechanism used to trigger end of game session. Defines how long, in milliseconds,
// between each tick in the example tick loop
const tickTime = 1000;

// Defines how to long to wait in Seconds before beginning early termination check in the
// example tick loop
const minimumElapsedTime = 120;

var session;                        // The Realtime server session object
var logger;                         // Log at appropriate level
via .info(), .warn(), .error(), .debug()
var startTime;                      // Records the time the process started
var activePlayers = 0;              // Records the number of connected players
var onProcessStartedCalled = false; // Record if onProcessStarted has been called
```
// Example custom op codes for user-defined messages
// Any positive op code number can be defined here. These should match your client code.
const OP_CODE_CUSTOM_OP1 = 111;
const OP_CODE_CUSTOM_OP1_REPLY = 112;
const OP_CODE_PLAYER_ACCEPTED = 113;
const OP_CODE_DISCONNECT_NOTIFICATION = 114;

// Example groups for user defined groups
// Any positive group number can be defined here. These should match your client code.
const RED_TEAM_GROUP = 1;
const BLUE_TEAM_GROUP = 2;

// Called when game server is initialized, passed server’s object of current session
function init(rtSession) {
  session = rtSession;
  logger = session.getLogger();
}

// On Process Started is called when the process has begun and we need to perform any
// bootstrapping. This is where the developer should insert any code to prepare
// the process to be able to host a game session, for example load some settings or set
// state
//
// Return true if the process has been appropriately prepared and it is okay to invoke the
// GameLift ProcessReady() call.
function onProcessStarted(args) {
  onProcessStartedCalled = true;
  logger.info("Starting process with args: " + args);
  logger.info("Ready to host games...");

  return true;
}

// Called when a new game session is started on the process
function onStartGameSession(gameSession) {
  // Complete any game session set-up

  // Set up an example tick loop to perform server initiated actions
  startTime = getTimeInS();
  tickLoop();
}

// Handle process termination if the process is being terminated by GameLift
// You do not need to call ProcessEnding here
function onProcessTerminate() {
  // Perform any clean up
}

// Return true if the process is healthy
function onHealthCheck() {
  return true;
}

// On Player Connect is called when a player has passed initial validation
// Return true if player should connect, false to reject
function onPlayerConnect(connectMsg) {
  // Perform any validation needed for connectMsg.payload, connectMsg.peerId
  return true;
}

// Called when a Player is accepted into the game
function onPlayerAccepted(player) {
  // This player was accepted -- let's send them a message
  const msg = session.newTextGameMessage(OP_CODE_PLAYER_ACCEPTED, player.peerId,
    "Peer " + player.peerId + " accepted");
  session.sendReliableMessage(msg, player.peerId);
}
activePlayers++;
}

// On Player Disconnect is called when a player has left or been forcibly terminated
// Is only called for players that actually connected to the server and not those rejected
// by validation
// This is called before the player is removed from the player list
function onPlayerDisconnect(peerId) {
    // send a message to each remaining player letting them know about the disconnect
    const outMessage = session.newTextGameMessage(OP_CODE_DISCONNECT_NOTIFICATION,
        session.getServerId(),
        "Peer " + peerId + " disconnected");
    session.getPlayers().forEach((player, playerId) => {
        if (playerId != peerId) {
            session.sendReliableMessage(outMessage, peerId);
        }
    });
    activePlayers--;
}

// Handle a message to the server
function onMessage(gameMessage) {
    switch (gameMessage.opCode) {
    case OP_CODE_CUSTOM_OP1: {
        // do operation 1 with gameMessage.payload for example sendToGroup
        const outMessage = session.newTextGameMessage(OP_CODE_CUSTOM_OP1_REPLY,
            session.getServerId(), gameMessage.payload);
        session.sendGroupMessage(outMessage, RED_TEAM_GROUP);
        break;
    }
    }
}

// Return true if the send should be allowed
function onSendToPlayer(gameMessage) {
    // This example rejects any payloads containing "Reject"
    return (!gameMessage.getPayloadAsText().includes("Reject"));
}

// Return true if the send to group should be allowed
// Use gameMessage.getPayloadAsText() to get the message contents
function onSendToGroup(gameMessage) {
    return true;
}

// Return true if the player is allowed to join the group
function onPlayerJoinGroup(groupId, peerId) {
    return true;
}

// Return true if the player is allowed to leave the group
function onPlayerLeaveGroup(groupId, peerId) {
    return true;
}

// A simple tick loop example
// Checks to see if a minimum amount of time has passed before seeing if the game has ended
async function tickLoop() {
    const elapsedTime = getTimeInS() - startTime;
    logger.info("Tick... " + elapsedTime + " activePlayers: " + activePlayers);

    // In Tick loop - see if all players have left early after a minimum period of time has passed
    // Call processEnding() to terminate the process and quit
    if ( (activePlayers == 0) && (elapsedTime > minimumElapsedTime)) {
        logger.info("All players disconnected. Ending game");
    }
Adding FlexMatch Matchmaking

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.

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 Setting Up Amazon GameLift FlexMatch Matchmakers (p. 138). For more information on how FlexMatch works, see How Amazon GameLift FlexMatch Works (p. 13).

Topics

- FlexMatch Integration Roadmap (p. 70)
- Add FlexMatch to a Game Client (p. 70)
- Add FlexMatch to a Game Server (p. 74)
- Backfill Existing Games with FlexMatch (p. 76)
FlexMatch Integration Roadmap

FlexMatch is available with the managed GameLift solutions 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. 141)
    - FlexMatch Rule Set Examples (p. 148)
  - **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 (p. 133)
- **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 Notification (p. 163)
- **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. 138)
  - Create a Matchmaking Configuration (p. 139)

- **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. 70)
- **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 Game Server (p. 74)
- **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. 76)

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

Add FlexMatch to a Game Client

This topic describes how to add FlexMatch matchmaking support to your game clients. FlexMatch is available with the managed GameLift solutions for custom game servers and Realtime Servers.

To learn more about FlexMatch and how to set up a custom matchmaker for your games, see these topics:

- How Amazon GameLift FlexMatch Works (p. 13)
- Setting Up Amazon GameLift FlexMatch Matchmakers (p. 138)
- FlexMatch Integration Roadmap (p. 70)
To enable FlexMatch on your game clients, you need to prepare your game client project and then add the following functionality:

- Request matchmaking for one or multiple players.
- Track the status of matchmaking requests.
- Request player acceptance for a proposed match.
- Once a game session is created for the new match, get player connection information and join the game.

**Prepare Client Service for Matchmaking**

We highly recommend that your game client make matchmaking requests through a client-side game service instead of directly. 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 For Client Services (p. 24) 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 tickets 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. 76).

**Request Matchmaking for Players**

Add code to your client service to create and manage matchmaking requests to a FlexMatch matchmaker.

**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. 138).
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 (p. 51).

- **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 the section called “Build a Rule Set” (p. 141).

- **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 Request Status

Add code to your client service to track the status of all matchmaking requests and respond as needed. There are a couple of options available for tracking status.

Event notifications
Set up notifications to track events that GameLift emits for matchmaking processes. This is the recommended method; it is simple to set up and an efficient use of resources. You can set up notifications either directly, by creating an SNS topic, or by using CloudWatch Events. For more information on setting up notifications, see Set up FlexMatch Event Notification (p. 163). 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 poll for status updates whenever 30 seconds has passed with no notification.

**Continuous polling**

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.

**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.

**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. If you're monitoring notifications, 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.

**Connect to a Match**

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

When a matchmaking request is completed, connection information is added to the matchmaking ticket. To 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 uses this information to connect directly to the game session that is hosting the match. A connection request for a matched game session 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).

**Sample StartMatchmaking Requests**

These 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.
Add FlexMatch to a Game Server

This topic describes how to add FlexMatch matchmaking support to your game server. FlexMatch is available with the managed GameLift solutions for custom game servers and Realtime Servers. To learn more about adding FlexMatch to your games, see these topics:
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 `onStartGameSession()` 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 (p. 44).

   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. 75).

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 (p. 44)). 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 (p. 44)). 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. 76).

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. 76).

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.

```json
{
    "matchId":"1111aaaa-22bb-33cc-44dd-5555eeee66ff",
    "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 match new players to existing matched game sessions. Although you can always add players to any game (see Join a Player to a Game Session (p. 50)), 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 Matchmaking Process (p. 14).

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

To add match backfill to your game, enable automatic backfill or manage backfill requests manually by adding code to your game server or game client.

**Turn on Automatic Backfill**

With automatic match backfill, GameLift automatically triggers a backfill request whenever a game session has an open player slot. 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. 139).

2. **Update game session with new matchmaker data.** Amazon GameLift updates your game server with match information using the Server SDK callback function `onUpdateGameSession` (see Prepare a Server Process (p. 42)). 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. 81).

3. **Turn off automatic backfill for a game session.** You can opt to stop automatic backfill at any point during an individual game session. For example, you might want backfill to occur only up to when gameplay actually starts or stop backfilling in the last few minutes of a game.

   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 Roadmap (p. 70).

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

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 matchmaking 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.

Version 77
Backfill Existing Games

- 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() (p. 238), add the onUpdateGameSession callback method name as a process parameter.

(p. 81)

As with other server functionality, a game server uses the Amazon GameLift Server SDK. This SDK is available in C++ and C#. For a general description of server APIs, see the Server API references (p. 213).

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(p. 241)
   - StopMatchBackfill(p. 242)

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() (p. 235). During the matchmaking process, tickets for new requests do not have a game session ID, while tickets for backfill requests do. The presence of a 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. 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. 75).
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 Prepare a Server Process (p. 42)). 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. 81).

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()` (p. 242). 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 Roadmap (p. 70).

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

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() (p. 238), add the `onUpdateGameSession` callback method name as a process parameter. (p. 81)

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. 71)), 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. 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. 75)). 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 Request Status (p. 72)) 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. 81).

**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):

"Success": The backfill request was successful.

"Cancelled": The backfill request was cancelled by the user or the system.

"Failed": The backfill request failed due to an error or timeout.

"Resubmitted": The backfill request was resubmitted due to a previous request being canceled or failed.

"MaxBackfills": The maximum number of backfill requests for a game session has been reached.

"InsufficientCapacity": The matchmaking service does not have enough capacity to complete the backfill request.

"PlayerCount": The number of players in the game session is not valid.

"Region": The region specified for the backfill request is not valid.

"GameSessionId": The game session ID specified for the backfill request is not valid.

"MaxSessions": The maximum number of sessions for a game session has been reached.

"MatchmakingConfiguration": The matchmaking configuration specified for the backfill request is not valid.

"PlayerCountMismatch": The number of players specified for the backfill request does not match the number of players in the game session.

"PlayerId": The player ID specified for the backfill request is not valid.

"PlayerData": The player data specified for the backfill request is not valid.

"PlayerLatency": The player latency data specified for the backfill request is not valid.

"PlayerSession": The player session specified for the backfill request is not valid.

"Region": The region specified for the backfill request is not valid.

"GameSessionId": The game session ID specified for the backfill request is not valid.

"MaxSessions": The maximum number of sessions for a game session has been reached.

"MatchmakingConfiguration": The matchmaking configuration specified for the backfill request is not valid.

"PlayerCount": The number of players in the game session is not valid.

"PlayerId": The player ID specified for the backfill request is not valid.

"PlayerData": The player data specified for the backfill request is not valid.

"PlayerLatency": The player latency data specified for the backfill request is not valid.

"PlayerSession": The player session specified for the backfill request is not valid.

"Region": The region specified for the backfill request is not valid.

"GameSessionId": The game session ID specified for the backfill request is not valid.

"MaxSessions": The maximum number of sessions for a game session has been reached.

"MatchmakingConfiguration": The matchmaking configuration specified for the backfill request is not valid.

"PlayerCount": The number of players in the game session is not valid.

"PlayerId": The player ID specified for the backfill request is not valid.

"PlayerData": The player data specified for the backfill request is not valid.

"PlayerLatency": The player latency data specified for the backfill request is not valid.

"PlayerSession": The player session specified for the backfill request is not valid.

"Region": The region specified for the backfill request is not valid.
• 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()` (p. 238), add the `onUpdateGameSession` callback method name as a process parameter.
Managing GameLift Hosting Resources

The topics in this section provide detailed help with setting up and managing your game server hosting resources for use with a managed GameLift solution. Whether you're deploying a fully custom game server or working with Realtime Servers, you need to allocate resources, configure them, and scale capacity to meet player demand.

**Tip**
Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using sample games (p. 29).

**Topics**
- About Amazon GameLift Hosting Resources (p. 83)
- Create Resources Using AWS CloudFormation (p. 84)
- Uploading GameLift Server Builds for Deployment (p. 91)
- Setting Up Amazon GameLift Fleets (p. 99)
- Scaling Amazon GameLift Fleet Capacity (p. 119)
- Using Multi-Region Queues (p. 128)
- Setting Up Amazon GameLift FlexMatch Matchmakers (p. 138)
- VPC Peering for Amazon GameLift (p. 165)

**About Amazon GameLift Hosting Resources**

This topic provides a brief overview of the key resources you work with when using the managed GameLift service to host your games. Learn more about how GameLift works in [How Amazon GameLift Works](p. 3).

The following diagram illustrates the basic structure of GameLift resources and how they relate to each other. As shown, you can structure a resource set to have one fleet or multiple fleets deployed with a single build or script.

**Build**
Your custom-built game server software that runs on GameLift and hosts game sessions for your players. A game build represents the set of files that run your game server on a particular operating system. You can have many different builds, such as for different flavors of your game. The game build must be integrated with the GameLift service. You upload game build files to the GameLift service in each region where you plan to deploy your game.
Script

Your configuration and custom game logic for use with Realtime Servers. Realtime Servers are provided by GameLift to use instead of providing a custom game server. You create a script, using JavaScript, to configure Realtime Servers for your game clients and add custom game logic as appropriate to host game sessions for your players. You upload the Realtime script to the GameLift service in each region where you plan to deploy your game.

Fleet

A collection of virtual hosting resources that run your game servers and host game sessions for your players. You create a fleet to deploy either a custom game build or a Realtime server script in a region, and you configure how you want the fleet to operate for your game. You set scaling rules to manage the fleet's size, which determines how many game sessions and players it can support. You can create more than one fleet for a build; for example, you might create a Spot fleet and an On-Demand fleet for a build. When deploying your game in multiple regions, you create a fleet in each region where you plan deploy your game.

Alias

Abstract identifier for a fleet. You can change the fleet it points to at any time. Since fleets are replaced whenever you update your custom game build, it is good practice to create an alias and use the alias wherever you need to reference a specific fleet (such as in a game session queue). When you replace the fleet and are ready for it to go live, switch the alias to point to the new fleet and redirect your players with no disruptions. You create an alias in the same region as the fleet it points to.

Game Session Queue

List of destinations (fleets or aliases) where game sessions can be hosted. A request for a new game session must specify where to place the session by providing either specific fleet ID or a game session queue ID. Queues, which can point to destinations across multiple regions, result in faster, more efficient game session placement. You can set your queue to enforce player latency limits to protect players from unacceptable game lag. Queues are required for games that use FlexMatch matchmaking or Spot fleets. You can create a game session queue in any region.

Matchmaking configuration

Your configuration for how FlexMatch should process matchmaking requests. Two key configuration settings are the matchmaking rule set and the game session queue. The matchmaking rule set tells GameLift how to evaluate potential players for a match. The game session queue tells GameLift where to host a new game session for the match. You can have more than one matchmaking configuration. Each request for a new match must specify the matchmaking configuration to use. You can create a matchmaking configuration in any region.

Matchmaking rule set

Your instructions for how to build the best possible matches for your game. The rule set defines the match's team structure and size, and contains rules for evaluating potential players. Rules can set requirements for individual players or for entire teams. A rule set can also set guidelines for when and how to relax the rules over time to minimize player wait times. A rule set is used by a matchmaking configuration. You create a matchmaking rule set in the same region as the matchmaking configuration that uses it.

Create Resources Using AWS CloudFormation

You can use AWS CloudFormation to manage your GameLift resources (p. 83). The GameLift console and CLI commands are useful tools to create and update individual resources. But with AWS CloudFormation you can manage an entire set of resources to support your game hosting. In AWS CloudFormation, you create a template that models each resource and then use the template to create your resources. To update resources, you make the changes to your template and use AWS CloudFormation to update the resources.
CloudFormation to implement the updates. You can organize your resources into logical groups, called stacks and stack sets.

Using AWS CloudFormation to maintain your GameLift hosting resources offers a more efficient way to manage sets of AWS resources. You can use version control to track template changes over time and coordinate updates made by multiple team members. You can also reuse templates. For example, when deploying a game across multiple Regions, you might use the same template to create identical resources in each Region. You can also use these templates to deploy the same sets of resources in another partition.

For more information about AWS CloudFormation, see the AWS CloudFormation User Guide. To view template information for GameLift resources, see the Amazon GameLift Resource Type Reference.

The following topics cover best practices with using AWS CloudFormation with GameLift and present some recommendations for structuring your resource templates.

**Best Practices**

For detailed guidance on using AWS CloudFormation, see the AWS CloudFormation Best Practices in the AWS CloudFormation User Guide. In addition, these best practices have special relevance with GameLift.

- **Consistently manage your resources using through AWS CloudFormation only.** This is a core AWS CloudFormation best practice, but it bears repeating. If you change your resources outside of AWS CloudFormation, such as using the GameLift console, GameLift API calls or CLI commands, your resources will get out of sync with your resource templates. This may result in unexpected consequences the next time you update your resources using the AWS CloudFormation templates.

- **Use AWS CloudFormation stacks and stack sets to efficiently manage multiple resources.**
  - Use stacks to manage groups of connected resources. AWS CloudFormation intelligently updates resources in a stack that reference each other based on whether resource properties are mutable. For example, suppose you have a stack that contains a build, a fleet that references the build, and an alias that references the fleet. In GameLift, the relationship between builds and fleets is immutable. If you update your template to replace a build, AWS CloudFormation also replaces the fleets that are connected to the replaced build. AWS CloudFormation then updates the existing aliases to point to the new fleets. For more information, see Working with Stacks in the AWS CloudFormation User Guide.
  - Use AWS CloudFormation stack sets if you're deploying identical stacks across multiple regions or AWS accounts. For more information, see Working with Stack Sets in the AWS CloudFormation User Guide.

- **If you are using Spot Instances, include an On-Demand Fleet as a back-up.** We recommend setting up your templates with two fleets in each region, one fleet with Spot Instances, and one fleet with On-Demand Instances. GameLift's FleetIQ feature ensures that game sessions are always placed first with viable Spot Instances. The On-Demand fleet acts as fallback in the event that the Spot Fleet is not available.

- **Group your Region-specific resources and global resources into separate stacks when you are managing resources in multiple Regions.** Some resources, such as GameLift fleets, can only reference other resources in the same Region. Other resources, such as GameLift queues, can reference resources in other regions. Placing them in separate stacks gives you more flexibility in where you place your global resources.

- **Place your global resources in close proximity to the services that use it.** When you are placing global resources, keep in mind how these resources are being accessed. Resources like queues and matchmaking configurations tend to receive a high volume of requests from specific sources, such as a backend service. By placing your resources in close proximity to the source of those requests, you minimize the request travel time and can improve overall performance.

- **Place your matchmaking configuration in the same Region as the game session queue that it uses.** Matchmaking configurations send requests for new game sessions to the queue, so placing these resources together also helps to optimize system performance.
• **Create a separate alias for each fleet in the stack.** Aliases make it much easier to transition player traffic when replacing game builds and fleets.

---

Using AWS CloudFormation Stacks

The following are recommended structures to use when setting up AWS CloudFormation stacks for GameLift-related resources. Your optimal stack structure varies depending on whether you are deploying your game in only one Region or across multiple Regions.

**Stacks for a Single Region**

To manage GameLift resources in a single Region, we recommend a two-stack structure:

- **Support stack** – This stack contains resources that your GameLift resources depend on. At a minimum, this stack should include the S3 bucket where you store your custom game server or Realtime script files. The stack should also include an IAM role that gives GameLift permission to retrieve your files from the S3 bucket when creating a GameLift build or script resource. This stack might also contain other AWS resources that are used with your game, such as DynamoDB tables, Amazon Redshift clusters, and Lambda functions.

- **GameLift stack** – This stack contains all of your GameLift resources, including the build or script, a set of fleets, aliases, and game session queue. AWS CloudFormation creates a build or script resource with files stored in the S3 bucket location and deploys the build or script to one or more fleet resources. Each fleet should have a corresponding alias. The game session queue references some or all of the fleet aliases. If you are using FlexMatch for matchmaking, this stack also contains a matchmaking configuration and rule set.

The diagram below illustrates a two-stack structure for deploying resources in a single AWS Region.
Stacks for Multiple Regions

When deploying your game in more than one Region, keep in mind how resources can interact across Regions. Some resources, such as GameLift fleets, can only reference other resources in the same Region. Other resources, such as a GameLift queue, are Region agnostic. To manage GameLift resources in multiple Regions, we recommend the following structure.

- **Regional support stacks** – These stacks contain resources that your GameLift resources depend on. This stack must include the S3 bucket where you store your custom game server or Realtime script files. It might also contain other AWS resources for your game, such as DynamoDB tables, Amazon Redshift clusters, and Lambda functions. Many of these resources are Region specific, so you must create them in every Region. GameLift also needs an IAM role that allows access to these support resources. Because an IAM role is Region agnostic, you only need one role resource, placed in any Region and referenced in all of the other support stacks.

- **Regional GameLift stacks** – This stack contains the GameLift resources that must exist in each region where your game is being deployed, including the build or script, a set of fleets, and aliases. AWS CloudFormation creates a build or script resource with files in an S3 bucket location, and deploys the build or script to one or more fleet resources. Each fleet should have a corresponding alias. The game session queue references some or all of the fleet aliases. You can maintain one template to describe this type of stack and use it to create identical sets of resources in every Region.

- **Global GameLift stack** – This stack contains your game session queue and matchmaking resources. These resources can be located in any Region and are usually placed in the same Region. The queue can reference fleets or aliases that are located in any Region. To place additional queues in different Regions, create additional global stacks.

The diagrams below illustrate a multistack structure for deploying resources in several AWS Regions. The first diagram shows a structure for a single game session queue. The second diagram shows a structure with multiple queues.
Updating Builds

GameLift builds are immutable, as is the relationship between a build and a fleet. As a result, when you update your hosting resources to use a new set of game build files, the following need to happen:

- Create a new build using the new set of files (replacement).
- Create a new set of fleets to deploy the new game build (replacement).
- Redirect aliases to point to the new fleets (update with no interruption).
Deploy Build Updates Automatically

When updating a stack containing related build, fleet and alias resources, the default AWS CloudFormation behavior is to automatically perform these steps in sequence. You trigger this update by first uploading the new build files to a new S3 location. Then you modify your AWS CloudFormation build template to point to the new S3 location. When you update your stack with the new S3 location, this triggers the following AWS CloudFormation sequence:

1. Retrieves the new files from S3, validates the files, and creates a new GameLift build.
2. Updates the build reference in the fleet template, which triggers new fleet creation.
3. After the new fleets are active, updates the fleet reference in the alias, which triggers the alias to update to target the new fleets.
4. Deletes the old fleet.
5. Deletes the old build.

If your game session queue uses fleet aliases, player traffic is automatically switched to the new fleets as soon as the aliases are updated. The old fleets are gradually drained of players as game sessions end. Auto-scaling handles the task of adding and removing instances from each set of fleets as player traffic fluctuates. Alternatively, you can specify an initial desired instance count to quickly ramp up for the switch and enable auto-scaling later.

You can also have AWS CloudFormation retain resources instead of deleting them. For more information, see RetainResources in the AWS CloudFormation API Reference.

Deploy Build Updates Manually

If you want to have more control over when new fleets go live for players, you have some options. You can choose to manage aliases manually using the GameLift console or the CLI. Alternatively, instead of updating your build template to replace the build and fleets, you can add a second set of build and fleet definitions to your template. When you update the template, AWS CloudFormation creates a second build resource and corresponding fleets. Since the existing resources are not replaced, they are not deleted, and the aliases remain pointing at original fleets.

The main advantage with this approach is that it gives you the flexibility. You can create separate resources for the new version of your build, test the new resources, and control when the new fleets go live to players. A potential drawback is that it requires twice as many resources in each Region for a brief period of time.

The following diagram illustrates this process.
How Rollbacks Work

When executing a resource update, if any step is not completed successfully, AWS CloudFormation automatically initiates a rollback. This process reverses each step in sequence, deleting the newly created resources.

If you need to manually trigger a rollback, change the build template's S3 location key back to the original location and update your stack. A new GameLift build and fleet are created, and the alias switches over to the new fleet after the fleet is active. If you are managing aliases separately, you need to switch them to point to the new fleets.

For more information about how to handle a rollback that fails or gets stuck, see Continue Rolling Back an Update in the AWS CloudFormation User Guide.

Uploading GameLift Server Builds for Deployment

Before setting up computing resources to host your Amazon GameLift-enabled multiplayer game, you first need to upload a game server build—or, when using Realtime Servers, a set of server script files—to the managed Amazon GameLift service. By uploading your files, you create a GameLift build or script resource.

Topics

- Upload a Custom Server Build to GameLift (p. 91)
- Upload a Realtime Servers Script to Amazon GameLift (p. 96)

Upload a Custom Server Build to GameLift

Once your game server has been integrated with GameLift, upload the build files to the managed GameLift service so that it can be deployed for game hosting. This topic covers how to package your game's build files, create an optional build install script, and then upload the files using either the AWS CLI or the AWS SDK.

Add a Build Install Script

Create an install script for the operating system of your game build:
• Windows: create a batch file named "install.bat".
• Linux: create a shell script file named "install.sh".

When creating an install script, keep in mind the following:

• The script cannot take any user input.
• A build is installed on a hosting server in the following locations. File directories in your build package are recreated.
  • For Windows fleets: C:\game
  • For Linux fleets: /local/game
• During the installation process, the run-as user has limited access to the instance file structure. It has full rights to the directory where your build files are installed. If your install script takes actions that require administrator permissions, you'll need to specify admin access (sudo for Linux, runas for Windows). Permission failures related to the install script generate an event message indicating a problem with the script.
• On Linux, common shell interpreter languages such as bash are supported. Add a shebang (such as #!/bin/bash) to the top of your install script. If you need to verify support for your preferred shell commands, you can remotely access an active Linux instance and open a shell prompt. Learn more at Remotely Access Fleet Instances (p. 117).

**Example scripts**

These examples illustrate common script usage for Windows and Linux.

**Windows**

This example install.bat file installs Visual C++ runtime components required for the game server and writes the results to a log file. The component file is included in the build package at the root.

```
vcredist_x64.exe /install /quiet /norestart /log c:\game\vcredist_2013_x64.log
```

**Linux**

This example install.sh file illustrates using bash in your install script and writing results to a log file.

```
#!/bin/bash
echo 'Hello World' > install.log
```

**Package Your Game Build Files**

Before uploading your GameLift-enabled game server to the GameLift service for hosting, you need to package the game build files into a build directory. This directory must include all components required to run your game servers and host game sessions, including the following:

• **Game server binaries** – The binary files required to run the game server. A build can include binaries for multiple game servers, as long as they are built to run on the same platform (see supported platforms (p. 23)).
• **Dependencies** – Any dependent files required by your game server executables to run. Examples include assets, configuration files, and dependent libraries.
• **Install script** – Script file to handle tasks that are required to fully install your game build onto GameLift hosting servers. This file must be placed at the root of the build directory. An install script is run once as part of fleet creation.
You can set up any application in your build, including your install script, to securely access your resources on other AWS services. Learn more about possible ways to do this in Access AWS Resources From Your Fleets (p. 45).

Once you've packaged your build files, make sure your game server can run on a clean installation of your target operating system (not one that's been used for development). This step helps ensure that you include all required dependencies in your package and that your install script is accurate.

**Note**
If you're storing your game build files in an Amazon S3 bucket for uploading, you need to package the build files into a .zip file. See instructions for uploading using this method in Create a Build with Files in Amazon S3 (p. 94).

**Create a GameLift Build**

When creating a build and uploading your files, you have a couple of options:

- Create a Build from a File Directory (p. 93). This is the simplest and most commonly used method.
- Create a Build with Files in Amazon S3 (p. 94). With this option, you can manage your build versions in S3 under your AWS account.

With both methods, a new Build resource is created with a unique build ID and other metadata. The build is placed in *Initialized* status. Once GameLift successfully acquires the game server files, the build moves to *Ready* status. At this point, you can deploy it by creating a new GameLift fleet (see Deploy a GameLift Fleet for a Custom Game Build (p. 104)).

When you create a fleet, you specify which build to deploy to the fleet. When GameLift sets up the new fleet, it downloads the build files to each fleet instance, and installs it based on the build install script (if one is provided). Build files are installed on the instances in the following locations:

- For Windows fleets: C:\game
- For Linux fleets: /local/game

**Create a Build from a File Directory**

To create a game build with packaged game server files stored in any location, including a local directory, use the AWS Command Line Interface (AWS CLI) command upload-build. This command creates a new build record in GameLift and uploads files from a location that you specify.

1. **Send an upload request.** In a command line window, type the following command and parameters.

   ```
   aws gamelift upload-build --operating-system [supported OS] --build-root [build path]
   --name [user-defined name of build] --build-version [user-defined build number] --region [region name]
   ```

   - **Build root** – The directory path of your build files.
   - **Operating system** – Specify the game server build's OS. When a new fleet is created for this build, fleet instances are configured with the appropriate OS. GameLift supports several Windows and Linux varieties. This parameter is optional. If an operating system is not specified, GameLift uses the default value (WINDOWS_2012). Once the build is created, this value cannot be updated later.
   - **Name** – Provide a descriptive name for the new build. Build name does not need to be unique, and you can update this value at any time by updating the build resource.
   - **Build version** – Use this optional field to specify version details for the build files. Since each new version of your game server requires a new build resource, this information can provide a valuable differentiator.
• Region – Identify the GameLift-supported region where you want to create your build. You must create the build in the region where you plan to deploy fleets. If you’re deploying your game in multiple regions, you must create a build in each region.

   **Note**
   If you work in multiple regions, it is always a good idea to check your current default region. In the AWS console, the current region is always displayed in the upper right corner, with a dropdown list of available regions to select from. If you’re using the AWS CLI, check your current default using the `configure get (aws configure get region)`. Use the command `configure set (aws configure set region [region name])` to change your default region.

In response to your upload request, the GameLift service provides upload progress, and on a successful upload returns the new build record ID. Upload time depends on the size of your game files and the connection speed.

**Examples:**

```bash
aws gamelift upload-build --operating-system AMAZON_LINUX --build-root ~/mygame --name "My Game Nightly Build" --build-version "build 255" --region us-west-2

aws gamelift upload-build --operating-system WINDOWS_2012 --build-root "C:\mygame" --name "My Game Nightly Build" --build-version "build 255" --region us-west-2
```

2. **Check build status.** View the new build record, including current status, using `describe-build` (or `DescribeBuild`). You can also view status on the GameLift console.

   In a command line window, type the following command and parameters.

   ```bash
   aws gamelift describe-build --build-id [build ID returned with the upload request] --region [region name]
   ```

   **Example:**

   ```bash
   aws gamelift describe-build --build-id "build-3333cccc-44dd-55ee-66ff-7777aaaa88bb" --region us-west-2
   ```

   In response to your request, the GameLift service returns the requested build record. This record contains a set of build metadata including status, size of the uploaded files, and a creation time stamp.

### Create a Build with Files in Amazon S3

To create a game build with packaged game server files that are stored in an Amazon S3 bucket under your AWS account, use the AWS CLI command `create-build`. This operation creates a new build in GameLift and acquires your build files from the Amazon S3 bucket that you specify.

1. **Store your build files in Amazon S3.** Create a .zip file containing the packaged build files and upload it to an Amazon S3 bucket under your AWS account. Take note of the bucket label and the file name; you'll need these when creating an GameLift build.

2. **Give GameLift access to your build files.** Follow the instructions at Set Up a Role for Amazon GameLift Access (p. 22) to create an IAM role. A role specifies which entities (such as the GameLift service) can assume the role and defines a set of permissions for limited access to your AWS resources. Once you've created the role, take note of the new role's Amazon Resource Name (ARN), which you'll need when creating a build.
3. **Send a request to create a new build.** Use the AWS CLI command `create-build` (or the AWS SDK operation `CreateBuild`) to create a new build record. Specify the S3 location where your build files are stored. In a command line window, type the following command and parameters.

```bash
aws gamelift create-build --operating-system [supported OS] --storage-location "Bucket=[S3 bucket label],Key=[Build zip file name],RoleArn=[Access role ARN]" --name [user-defined name of build] --build-version [user-defined build number] --region [region name]
```

- **Build root** – The directory path of your build files.
- **Operating system** – Specify the game server build's OS. When a new fleet is created for this build, fleet instances are configured with the appropriate OS. GameLift supports several Windows and Linux varieties. This parameter is optional. If an operating system is not specified, GameLift uses the default value (WINDOWS_2012). Once the build is created, this value cannot be updated later.
- **Name** – Provide a descriptive name for the new build. Build name does not need to be unique, and you can update this value at any time by updating the build resource.
- **Build version** – Use this optional field to specify version details for the build files. Since each new version of your game server requires a new build resource, this information can provide a valuable differentiator.
- **S3 location**
  - **Bucket** – Name of the S3 bucket that contains your build. Example: "my_build_files".
  - **Key** – Name of the .zip file that contains your build files. Example: "mygame_build_7.0.1, 7.0.2".
- **Role ARN** – ARN assigned to the IAM role that you created. Example: "arn:aws:iam::111122223333:role/GameLiftAccess".
- **Region** – Identify the GameLift-supported region where you want to create your build. You must create the build in the region where you plan to deploy fleets. If you're deploying your game in multiple regions, you must create a build in each region.

**Note**

If you work in multiple regions, it is always a good idea to check your current default region. In the AWS console, the current region is always displayed in the upper right corner, with a dropdown list of available regions to select from. If you're using the AWS CLI, check your current default using the `configure get` (aws configure get region). Use the command `configure set` (aws configure set region [region name]) to change your default region.

**Example:**

```bash
aws gamelift create-build --operating-system WINDOWS_2012 --storage-location "Bucket=my_game_build_files,Key=mygame_build_101.zip,RoleArn=arn:aws:iam::111122223333:role/gamelift" --name "My Game Nightly Build" --build-version "build 101" --region us-west-2
```

In response to your request, the GameLift service returns the newly created build record, including the build's current status.

**Update Your Build Files**

Once an GameLift build has been created, the build files associated with it cannot be changed. Instead, you must create a new GameLift build for each new set of files. If you provide build files using the `upload-build` command, you don't need to do anything special because GameLift automatically creates a new build record for each request. If you provide build files using the `create-build`
command, upload a new build .zip file with a different name to Amazon S3 and create a build by referencing the new file name.

Try these tips for deploying updated builds:

- **Use queues and swap out fleets as needed.** When setting up your game client with GameLift, specify a queue instead of a fleet. With queues, you can create new fleets to run your new build, then add the new fleets to your queue and remove the old fleets. For details, see Using Multi-Region Queues (p. 128).
- **Use aliases to silently transfer players to a new game build.** When integrating your game client with GameLift, specify a fleet alias instead of a fleet ID. With aliases, you can move players to a new build in just three steps: (1) Create the new build. (2) Create a fleet to deploy the new build. (3) Change the alias target from the old fleet to the new fleet. For more information, see Add an Alias to a Amazon GameLift Fleet (p. 113).
- **Set up automated build updates.** Follow the GameTech blog post Automating Deployments to GameLift, with sample scripts, to incorporate GameLift deployments into your build system.

### Upload a Realtime Servers Script to Amazon GameLift

When you have completed a Realtime server script for your game, upload the script files to Amazon GameLift for deployment with your Realtime Servers. This is done by creating an Amazon GameLift script record and providing the script files. You have two options when providing the script files:

- Create a script with a zip file that is stored on a local directory.
- Create a script with a zip file that is stored in an Amazon S3 bucket.

When you create a new script, Amazon GameLift assigns a unique script ID (example: `script-1111aaaa-22bb-33cc-44dd-5555eeee66ff`) and adds the creation time stamp, uploaded file size, and other metadata. Upload time depends on the size of your script files and the connection speed.

At this point, the script can be deployed with a new Amazon GameLift Realtime Servers fleet. Amazon GameLift installs your server script onto each instance in the fleet, placing the script files at the following location: `/local/game`.

To troubleshoot fleet activation problems that might be related to the server script, see Debug Fleet Issues (p. 114).

### Package Your Script Files

Your server script can include one or multiple files. For uploading, combine all files that your JavaScript depends on to run into a single zip file.

### Upload Local Script Files

You can use either the Amazon GameLift console or the AWS CLI tool to create a script.

**GameLift Console**

1. Open the Amazon GameLift console at [https://console.aws.amazon.com/gamelift/](https://console.aws.amazon.com/gamelift/).
2. Use the GameLift main menu to open the Scripts: Create script page to access the script creation form.
3. **In Script configuration**, enter a script name and version information. Because the content of a script can be updated, version data can be helpful in tracking the updates.

4. **In Script code**, choose the **Script type** “Zip file”. This option lets you specify a zip file that is stored in a local directory.

5. Browse for the zip file that contains your script and select it.

6. Once you finished defining the new script record, click **Submit**. Amazon GameLift assigns an ID to the new script and begins uploading the designated zip file. You can view the new script record, including status, on the console's **Scripts** page.

**AWS CLI**

To create a script with the AWS CLI, open a command line window and use the `create-script` command to define the new script and upload your server script files. See complete documentation on this command in the [AWS CLI Command Reference](https://aws.amazon.com/documentation/cli/). Get and install the AWS Command Line Interface tool.

**To create a script**

1. Place the zip file into a directory that you can make calls to the AWS CLI.
2. In a command line window, switch to the directory where the zip file is located.
3. Enter the `create-script` command and parameters. For the `-zip-file` parameter, be sure to prepend the string "fileb:/" to the name of the zip file. It identifies the file as binary and ensures that the compressed content will processed correctly.

   ```bash
   aws gamelift create-script --name [user-defined name of script] --script-version [user-defined version info] --zip-file fileb://[name of zip file] --region [region name]
   ```

   In response to your request, the Amazon GameLift service returns the new script object.

   **Examples:**

   ```bash
   aws gamelift create-script --name My_Realtime_Server_Script_1 --script-version 1.0.0 --zip-file fileb://myrealtime_script_1.0.0.zip --region us-west-2
   ```

   You can view the new script by calling `describe-script` or by viewing it in the Amazon GameLift console.

**Upload Script Files in Amazon S3**

You can opt to store your script files in an Amazon S3 bucket and upload them to Amazon GameLift from there. When you create your script, you specify the S3 bucket location and Amazon GameLift acquires your script files directly from S3.

**To create a script**

1. **Store your script files in an Amazon S3 bucket.** Upload the zip file containing your server script into an Amazon S3 bucket in an AWS account that you control. Take note of the bucket name and the file name (also called the "key"); you'll need these when creating an Amazon GameLift script.
2. **Give Amazon GameLift access to your script files.** Follow the instructions at [Set Up a Role for Amazon GameLift Access (p. 22)](https://aws.amazon.com/documentation/gamelift/) to create an IAM role that allows Amazon GameLift to access the Amazon S3 bucket containing your server script. Once you've created the role, take note of the new role's Amazon Resource Name (ARN), which you'll need when creating a script.
3.  **Create a script.** Use either the Console or the AWS CLI to create a new script record.

**GameLift Console**

1.  Use the GameLift main menu to open the **Scripts: Create script** page to access the script creation form.
2.  In **Script configuration**, enter a script name and version information. Because the content of a script can be updated, version data can be helpful in tracking the updates.
3.  In **Script code**, choose the **Script type** "User storage". This option lets you specify the S3 bucket containing your script file.
4.  Enter the storage location information for your S3 bucket:
   - S3 bucket – The bucket name.
   - S3 key – The name of the file (zipped file containing your server script) in the bucket.
   - S3 role ARN – The ARN value for the IAM role that your created in Step 2.
   - S3 object version – (optional) A specific version number for the S3 file. This is used only when the S3 bucket has object versioning turned on AND when you want to specify a version other than the latest one.
5.  Once you finished defining the new script record, click **Submit**. Amazon GameLift assigns an ID to the new script and begins uploading the designated zip file. You can view the new script record, including status, on the console's **Scripts** page.

**AWS CLI**

Use the `create-script` command to define the new script and upload your server script files. See complete documentation on this command in the AWS CLI Command Reference. Get and install the AWS Command Line Interface tool.

1.  Open a command line window and switch to a directory where you can use the AWS CLI tool.
2.  Enter the `create-script` command with the following parameters: `--name`, `--script-version`, and `--storage-location`. The storage location parameter specifies the Amazon S3 bucket location of your script files.

```bash
aws gamelift create-script --name [user-defined name of script] --script-version [user-defined version info] --storage-location "Bucket=[S3 bucket name],Key=[name of zip file in S3 bucket],RoleArn=[Access role ARN]" --region [region name]
```

In response to your request, the Amazon GameLift service returns the new script object.

**Examples:**

```bash
aws gamelift create-script --name My_Realtime_Server_Script_1 --script-version 1.0.0 --storage-location "Bucket=gamelift-script,Key=myrealtime_script_1.0.0.zip,RoleArn=arn:aws:iam::123456789012:role/S3Access" --region us-west-2
```

You can view the new script by calling `describe-script` or by viewing it in the Amazon GameLift console.

**Update Script Files**

You can update either the metadata or the script content at any time using either the GameLift Console or the AWS CLI command `update-script`. When you update script content, the new script is used for all
new game sessions, but does not affect currently running game sessions that were created before the
script update.

If the script is stored in an Amazon S3 bucket under your account, you can update the script content by
uploading a new zip file to the bucket. If the script is provided as a local zip file, you must upload the
new file to GameLift.

You also can change the location where script content is located without having to create a new script
record.

## Setting Up Amazon GameLift Fleets

This section provides detailed help with designing, building, and managing your fleets. Start with Design
a Amazon GameLift Fleet for Your Game (p. 99) to learn about the various options for creating a fleet.

Fleets are your hosting resources in the form of a set of EC2 instances. To host game servers on Amazon
GameLift, deploy a fleet with either a custom game server or Realtime Servers. The size of a fleet
depends on the number of instances you give it, and you can adjust fleet size to meet player demand
either manually or by auto-scaling.

Most games in production require more than one fleet. You need multiple fleets if, for example, you want
to host players in more than one region, or if you have two or more versions of your game server build or
script (such as free and premium versions).

**Tip**

Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using
sample games (p. 29).

Once you have a fleet up and running, you can create a fleet alias (p. 113), add the fleet to a game
session queue (p. 128), and manage fleet capacity (p. 119).

**Topics**

- Design a Amazon GameLift Fleet for Your Game (p. 99)
- Deploy a GameLift Fleet for a Custom Game Build (p. 104)
- Deploy a Realtime Servers Fleet (p. 108)
- Manage Fleet Records (p. 111)
- Add an Alias to a Amazon GameLift Fleet (p. 113)
- Debug Fleet Issues (p. 114)
- Remotely Access Fleet Instances (p. 117)

## Design a Amazon GameLift Fleet for Your Game

The topics in this section provide help with designing a fleet of computing resources that is well-
suited to your game. Get help with selecting the right hosting resources and deciding how to use those
resources by creating a runtime configuration.

**Tip**

Learn more about ways to explore Amazon GameLift features, including Realtime Servers, using
sample games (p. 29).

**Topics**

- Choose Computing Resources (p. 100)
- Run Multiple Processes on a Fleet (p. 101)
- Spot Fleet Integration Guide (p. 103)
Choose Computing Resources

Amazon GameLift uses Amazon Elastic Compute Cloud (Amazon EC2) resources, called instances, to deploy your game servers and host game sessions for your players. When setting up a new fleet, you decide what type of instances your game needs and how to run game server processes on them (via the runtime configuration). Once a fleet is active and ready to host game sessions, you can add or remove instances at any time to accommodate more or fewer players. All instances in a fleet use the same type of resources and the same runtime configuration. You can edit a fleet’s runtime configuration, but the type of resources cannot be changed.

When choosing resources for a fleet, you'll need to consider several factors, including game operating system, instance type (the computing hardware), and whether to use on-demand or spot instances. Each of these issues is discussed further in this topic. Keep in mind that hosting costs with Amazon GameLift primarily depend on the type of resources you use. Learn more about Amazon GameLift pricing.

Operating Systems

Amazon GameLift supports game server builds that run on either Microsoft Windows or Amazon Linux (see supported game server operating systems (p. 23)). When uploading a game build to Amazon GameLift, you specify the operating system for the game. When you create a fleet to deploy the game build, Amazon GameLift automatically sets up instances with the build's operating system.

The cost of resources depends on the operating system in use. Learn more about the resources available for the supported operating systems:

- Microsoft Windows
- Amazon Linux

Instance Types

A fleet’s instance type determines the kind of hardware that will be used for every instance in the fleet. Instance types offer different combinations of computing power, memory, storage, and networking capabilities. With Amazon GameLift you have a wide range of instance type options to choose from. View a list of available instance types or open the Service limits page in the Amazon GameLift console to view instance types, current usage, and usage limits. To learn more about the capabilities of each instance type, see Amazon EC2 Instance Types. Note that the instance types offered may vary depending on the region.

When choosing an instance type for your game, consider the following: (1) the computing requirements of your game server build, and (2) the number of server processes that you plan to run on each instance. You may be able to run multiple server processes on each instance by using a larger instance type, which can reduce the number of instances you need to meet player demand. However, larger instance types also cost more. Learn more about running multiple processes on a fleet (p. 101).

On-Demand versus Spot Instances

When creating a new fleet, you designate the fleet type as using either On-demand or Spot instances. On-demand and Spot instances offer exactly the same hardware and performance, based on the instance type chosen, and are configured in exactly the same way. They differ in availability and in cost.

On-demand instances

On-demand instances are simply that: you request an instance and it is created for you. You can always acquire an On-demand instance when you need it and you can keep it as long as you want. On-demand instances have a fixed cost—you pay for the amount of time that you use them and there are no long-term commitments.
Spot instances

Spot instances can offer a highly cost-efficient alternative to on-demand instances. Spot instances take advantage of currently unused AWS computing capacity, which is what can make them less expensive but also less available. When using Spot instances, it is important to be aware of two key facts: (1) Spot prices fluctuate based on the supply and demand for each instance type in each region, and (2) Spot instances can potentially be interrupted by AWS with a two-minute notification when AWS needs the capacity back.

Amazon GameLift FleetIQ, however, significantly mitigates the chance of interruptions, allowing you to achieve cost savings while maintaining high game server availability. FleetIQ is responsible for finding the best available resources for each new game session. When a fleet has Spot instances, FleetIQ prioritizes using instances with the highest cost savings and historically low interruption rates. So, if your fleet includes a mixture of t2 and c4 Spot instances, and demand for t2 instances has recently been high, FleetIQ evaluates the t2 instance's interruption risk and may opt to use a larger c4 instance.

You can evaluate FleetIQ’s performance using a set of queue metrics, as well as instance-specific metrics on spot instances. Learn more about Amazon GameLift metrics (p. 178). You can also view pricing history for any instance type in the Amazon GameLift console. The Spot history page graphs On-demand and Spot pricing and calculates the relative cost savings with Spot instances. Use the controls to select an instance type, operating system, and a time range.

Learn more about how to use Spot instances in the Spot Fleet Integration Guide (p. 103).

Instance Service Quotas

AWS limits the total number of Amazon EC2 instances (On-Demand or Spot) that you can have in an AWS account. In addition, each instance type has a maximum limit, which varies by region. You can extend these quotas by request.

You can access information on service quotas:

- AWS Endpoints and Quotas lists general quota information for GameLift, as well as other AWS services.
- Look up specific quota values for instance types per region in the Amazon GameLift console. Go to the Service Limits page and select a region to view quotas for the available instance types. This page also displays current usage values for each instance type in the region.
- Retrieve quota values for instance types per region with the AWS CLI command describe-ec2-instance-limits. This action also returns the number for instances currently active in the region.

Run Multiple Processes on a Fleet

This topic provides additional information on how to set up a fleet’s runtime configuration to run multiple processes per instance. Depending on how you configure your fleet, running multiple processes can give you greater control over your use of Amazon GameLift resources, improving efficiency and potentially reducing overall hosting costs.

Optimizing for Multiple Processes

At a minimum, you must do the following to enable multiple processes:

- Create a build (p. 91) that contains all of the game server executables that you want to deploy to a fleet and upload it to Amazon GameLift. All game servers in a build must run on the same platform and be integrated with Amazon GameLift using the Amazon GameLift Server SDK for C++, version 3.0.7 or later.
- Create a runtime configuration with one or more server process configurations and multiple concurrent processes.
Game clients connecting to games hosted on this fleet must be integrated using the AWS SDK, version 2016-08-04 or later.

In addition, implement the following in your game servers to optimize fleet performance:

- Handle server process shutdown scenarios to ensure that Amazon GameLift can recycle processes efficiently. If you don't do this, server processes can't be shut down until they fail.
- Add a shutdown procedure to your game server code, ending with the server API call to `ProcessEnding()`.
- Implement the callback function `OnProcessTerminate()` in your game server code to gracefully handle termination requests from Amazon GameLift.
- Make sure that "unhealthy" server processes are shut down and relaunched quickly by defining what "healthy" and "unhealthy" mean and reporting this status back to Amazon GameLift. You do this by implementing the `OnHealthCheck()` callback function in your game server code. Amazon GameLift automatically shuts down server processes that are reported unhealthy for three consecutive minutes. If you don't implement `OnHealthCheck()`, Amazon GameLift assumes a server process is healthy unless it fails to respond. As a result, poorly performing server processes can continue to exist, using up resources until they finally fail.

How a Fleet Manages Multiple Processes

Amazon GameLift uses a fleet runtime configuration to manage what processes to maintain on each instance in the fleet. A runtime configuration is actually made up of one or multiple server process configurations, each of which identifies the following:

- The path and file name of a server executable in the game build deployed on the fleet
- (Optional) Parameters to pass to the server process on launch
- The number of this server process to maintain concurrently on the instance

When an instance is started in the fleet, the instance immediately begins launching server processes called for in the runtime configuration. Server process launches on an instance are staggered by a few seconds, so depending on the total number of server processes configured for an instance, it may take a few minutes to achieve full capacity.

Over time, server processes end, either by self-terminating (calling the Server SDK `ProcessEnding()`), or by being terminated by Amazon GameLift. An instance regularly checks that it is running the number and type of server processes specified in the runtime configuration. If not, the instance automatically launches server processes as needed to meet the runtime configuration requirements. As a result, as server processes end, their resources are continually recycled to support new server processes, and instances generally maintain the expected complement of active server processes.

You can change a fleet's runtime configuration at any time by adding, changing, or removing server process configurations. Here's how Amazon GameLift adopts runtime configuration changes.

1. Before an instance checks that it is running the correct type and number of server processes, it first requests the latest version of the fleet's runtime configuration from the Amazon GameLift service. If you've changed the runtime configuration, the instance acquires the new version and implements it.
2. The instance checks its active processes against the current runtime configuration and handles discrepancies as follows:
   - The updated runtime configuration removes a server process type. Active server processes that no longer match the runtime configuration continue to run until they end.
   - The updated runtime configuration decreases the number of concurrent processes for a server process type. Excess server processes of that type continue to run until they end.
• The updated runtime configuration adds a new server process type or increases the concurrent processes setting for an existing type. New server processes are launched immediately to match the updated runtime configuration, unless the instance is already running the maximum number of server processes. In this case, new server processes are launched only when existing processes end.

Choosing the Number of Processes per Instance

There are effectively three limits to keep in mind when deciding on the number of concurrent processes:

• Amazon GameLift limits each instance to a maximum number of concurrent processes. Whether your runtime configuration has one or multiple server process configurations specified, the sum of all concurrent processes for a fleet's server process configurations can't exceed this limit.

• The Amazon EC2 instance type that you choose may limit the number of processes that can run concurrently while offering acceptable performance levels. You need to test different configurations for your game to find the optimal number of processes for your preferred instance type. Factors that may affect your choice include the resource requirements of your game server, the number of players to be hosted in each game session, and player performance expectations.

• When changing a fleet's runtime configuration, keep in mind that Amazon GameLift will never run more concurrent processes than the total number configured. This means that the transition from old runtime configuration to new may happen gradually, with new processes starting only as existing processes end. Here's an example: You have a fleet that is configured to run 10 concurrent processes of your server executable, `myGame.exe`, with launch parameters set to `"-loglevel=1"`. You update the configuration to continue running 10 concurrent processes of `myGame.exe` but change the launch parameters to `"-loglevel=4"`. Since instances in the fleet are already running 10 processes, Amazon GameLift waits to start a process with the new launch parameters until a process with the old launch parameters ends.

Spot Fleet Integration Guide

When setting up fleets of hosting resources, you have the option of using both spot instances and on-demand instances. Learn more about how Amazon GameLift uses spot instances in On-Demand versus Spot Instances (p. 100). If you choose to use spot fleets, you'll need to make a few adjustments to your game integration.

Are you using FlexMatch for matchmaking? You can use spot fleets with your existing queues for matchmaking placements.

To set up game and hosting resources for spot fleets:

1. **Design a queue with a "spot-optimized" configuration.** Managing fleet usage with a queue is always a best practice, but it's required when using spot instances. For this step, you'll need to identify which regions to cover in the queue, select the right instance types (hardware) for your game, and decide how you want FleetIQ to prioritize game session placement. As a best practice, each region in your queue should have more than one Spot fleet and one On-Demand fleet. The spot fleets should use different instance types, ideally in the same family (`c4.large`, `c4.xlarge`, etc.). This design increases the resiliency of your game servers. See Design a Queue for Spot Instances (p. 131) for help designing a spot-optimized queue.

   Once you've created the fleets for your queue, you can create the queue, add fleet destinations, and set fleet prioritization. See Create a Queue (p. 133) for help creating and configuring the new queue.

2. **Create the fleets for your spot-optimized queue.** Before creating the queue, you need to create the fleets for it. Spot fleets and on-demand fleets can have identical configurations, with the only difference being the fleet type designation. See Deploy a GameLift Fleet for a Custom Game Build (p. 104) for help creating and configuring new fleets.
Note
We recommend that you include the fleet type (spot or on-demand) in the fleet name. This will make it easy to identify your spot fleets at a glance in a list of fleet names, such as when adding fleets to a queue.

3. **Enable your game client or client service to place new game sessions using the spot-optimized queue.** If you're not already doing so, update your game client code to use game session placement with a queue rather than creating game sessions on a single specific fleet. By using the queue to place game sessions, you're enabling FleetIQ to select the best available fleet based on cost, interruption rate, and player latency. You're also providing fallback hosting resources in case a preferred fleet is temporarily not available for use. For help implementing game session placements in your game client, see [Create Game Sessions](p. 49).

4. **Enable your game server to handle a spot interruption.** Spot instances can be interrupted with two minutes’ notification when AWS needs the capacity back. You'll want your game server to handle an interruption gracefully, if it happens, and minimize the player impact. When AWS is about to reclaim a spot instance, it sends termination notification up to two minutes before, which Amazon GameLift passes on to all affected server processes. This is done by invoking the Server SDK callback function `onProcessTerminate()`. Implementing this callback function is always a best practice, but it is particularly important when using spot instances. The implementation of this callback can take action to either gracefully end the game sessions or find a way to move the game sessions and players to a new placement. See [Shut Down a Server Process](p. 45) for help implementing `onProcessTerminate()`.

Note
AWS makes every effort to provide the notification as soon as an instance is chosen for termination, but there is the possibility that the spot instance will be terminated before the warning arrives. Your game server should be prepared to handle unexpected interruptions.

5. **Evaluate the performance of your spot fleets and queues.** Once the new queue is actively placing new game sessions, use Amazon GameLift metrics to evaluate performance. Key metrics include:

   - **Interruption rate** – Track the number and frequency of spot-related interruptions for instances and game sessions in a spot fleet. These fleet metrics (InstanceInterruptions and GameSessionInterruptions) can be viewed in the Amazon GameLift console or by using Amazon CloudWatch (see [Amazon GameLift Metrics for Fleets](p. 178)). If you aggregate fleet metrics in a metrics group, you can also view interruptions by instance type and operating system in CloudWatch. Game sessions that were terminated for spot-related reasons have a status of “TERMINATED” and a status reason of “INTERRUPTED”.
   
   - **Queue effectiveness** – Track metrics for your queues, including placement success rates, average wait time, and queue depth to verify that the use of spot fleets has no impact on queue performance. Queue metrics can be viewed in the Amazon GameLift console or by using Amazon CloudWatch.
   
   - **Fleet utilization** – Monitor usage rates for your fleets, including data on instances, game sessions and player sessions. Usage for your on-demand fleets can be an indicator that FleetIQ is choosing to avoid risky placements into spot fleets and falling back to the on-demand fleets. Fleet utilization metrics can be viewed in the Amazon GameLift console or by using Amazon CloudWatch.

### Deploy a GameLift Fleet for a Custom Game Build

If you're using Realtime Servers for your game, see [Deploy a Realtime Servers Fleet](p. 108).

You can create and deploy a new fleet to host game servers for any game build that has been uploaded to the Amazon GameLift service and is in a **Ready** status.

**Create a fleet**

Use either the [Amazon GameLift console](http://console.amazon.com) or the AWS Command Line Interface (CLI) to create a fleet.
After you create a new fleet, the fleet's status passes through several stages as the fleet is deployed and game servers installed and started up. Once the fleet reaches ACTIVE status, it is ready to host game sessions. For help with fleet creation issues, see Debug Fleet Issues (p. 114).

Console

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the Builds page, find the build that you want to create a fleet for and verify that its status is Ready. Select the build (use the option button to the left of the build status) and click Create fleet from build.
3. On the Create fleet page, enter the Fleet Details:
   - Name – Create a meaningful fleet name so you can easily identify it in a list and in metrics.
   - Description – (Optional) Add a short description for this fleet to further aid identification.
   - Fleet type – Choose whether to use on-demand or spot instances for this fleet. Learn more about fleet types in Choose Computing Resources (p. 100).
   - Metric group – (Optional) Enter the name of a new or existing fleet metric group. When using Amazon CloudWatch to track your Amazon GameLift metrics, you can aggregate the metrics for multiple fleets by adding them to the same metric group.
   - Instance role ARN – (Optional) Enter the ARN value for an IAM role that you want to associated with this fleet. This setting allows all instances in the fleet to assume the role, which extends access to a defined set of AWS services. Learn more about how to Access AWS Resources From Your Fleets (p. 45).
   - Certificate type – Choose whether to have GameLift generate a TLS certificate for the fleet. You can use a fleet TLS certificate to have your game client authenticate a game server when connecting, and encrypt all client/server communication. For each instance in a TLS-enabled fleet, GameLift also creates a new DNS entry with the certificate. Use these resources to set up authentication and encryption for your game. Once the fleet is created, you cannot change the certificate type.
   - Binary type – Select the binary type "Build槌.
   - Build – If you used the Create fleet from build feature, the build information, including name, ID and operating system, is automatically filled in. Otherwise, select a valid build from the dropdown list.
4. Instance type. Select an Amazon EC2 instance type from the list. The instance types listed vary depending on several factors, including the current region, the operating system of the selected game build, and the fleet type (on-demand or spot). Learn more about choosing an instance type in Choose Computing Resources (p. 100). Once the fleet is created, you cannot change the instance type.
5. Process management. Configure how you want server processes to run on each instance.
   a. Server process allocation:
      Specify the type and number of game server processes you want to run on each instance. Each fleet must have at least one server process configuration defined and can have multiple configurations. For example, if your game build has multiple server executables, you must have a configuration for each executable.
      - Launch path – Type the path to the game executable in your build. All launch paths must start with the game server location, which varies based on the operating system in use. On Windows instances, game servers are built to the path C:\game. On Linux instances, game servers are built to /local/game, so all launch paths must start with this location. Examples: C:\game\MyGame\server.exe or /local/game/MyGame/server.exe.
      - Launch parameters – (Optional) You can pass information to your game executable at launch time. Type the information as a set of command line parameters here. Example: +sv_port 33435 +start_lobby.
• **Concurrent processes** – Indicate how many server processes with this configuration to run concurrently on each instance in the fleet. Check the Amazon GameLift limits on number of concurrent server processes; they depend on which SDK your game server uses.

Once you enter a server process configuration, click the green checkmark button to save the configuration. To add additional server process configurations, click **Add configuration**.

Limits on concurrent server processes per instance apply to the total of concurrent processes for all configurations. If you're limited to one process, you can have only one configuration, and concurrent processes must be set to 1. If you configure the fleet to exceed the limit, the fleet cannot activate.

The collection of server process configurations is called the fleet's runtime configuration. It describes all server processes that will be running on each instance in this fleet at any given time.

b. **Game session activation:**

Set the following limits to determine how new game sessions are activated on the instances in this fleet:

• **Max concurrent game session activation** – Limit the number of game sessions on an instance that can be activating at the same time. This limit is useful when launching multiple new game sessions may have an impact on the performance of other game sessions running on the instance.

• **New activation timeout** – This setting limits the amount of time Amazon GameLift allows for a new game session activate. If the game session does not complete activation and move to status ACTIVE, the game session activation is terminated.

6. **EC2 port settings.** Click **Add port settings** to define access permissions for inbound traffic connecting to server processes deployed on this fleet. You can create multiple port settings for a fleet. At least one port setting must be set for the fleet before access is allowed. If you don't specify port settings at this time, you can edit the fleet later.

- **Port range** – Specify a range of port numbers that your game servers can use to allow inbound connections. A port range must use the format nnnnn[-nnnnn], with values between 1025 and 60000. Example: **1500** or **1500–20000**.

- **Protocol** – Select the type of communication protocol for the fleet to use.

- **IP address range** – Specify a range of IP addresses valid for instances in this fleet. Use CIDR notation. Example: **0.0.0.0/0** (This example allows access to anyone trying to connect.)

7. In the **Protection policy** section, choose whether to apply game session protection to instances in this fleet. Instances with protection are not terminated during a scale down event if they are hosting an active game session. You can also set protection for individual game sessions. Once the fleet is created, you can edit the fleet to change the fleet-wide protection policy.

8. Once you've finished setting the configuration for your new fleet, click **Initialize fleet**. Amazon GameLift assigns an ID to the new fleet and begins the fleet activation process. You can track the new fleet's status on the **Fleets** page.

You can update the fleet's metadata and configuration at any time, regardless of fleet status (see **Manage Fleet Records** (p. 111)). You can update fleet capacity only once the fleet has reached **ACTIVE** status (see **Scaling Amazon GameLift Fleet Capacity** (p. 119)).
AWS CLI

To create a fleet with the AWS CLI, open a command line window and use the `create-fleet` command to define a new fleet. See complete documentation on this command in the AWS CLI Command Reference. Get and install the AWS Command Line Interface tool.

The example `create-fleet` request shown below creates a new fleet with the following characteristics:

- The fleet will use c5.large on-demand instances with the operating system required for the selected game build.
- It will deploy the specified game server build, which must be in a **Ready** status.
- TLS certificate generation is enabled.
- Each instance in the fleet will run ten identical processes of the game server concurrently, enabling each instance to host up to ten game sessions simultaneously.
- On each instance, Amazon GameLift will allow only two new game sessions to be activating at the same time. It will also terminate any activating game session if it is not ready to host players within 300 seconds.
- All game sessions hosted on instances in this fleet have game session protection turned on. It can be turned off for individual game sessions.
- Individual players can create three new game sessions within a 15-minute period.
- Each game session hosted on this fleet will have a connection point that falls within the specified IP address and port ranges.
- Metrics for this fleet will be added to the EMEAfleets metric group, which (in this example) combines metrics for all fleets in EMEA regions.

```bash
$ aws gamelift create-fleet
--name "SampleFleet123"
--description "The sample test fleet"
--ec2-instance-type "c5.large"
--fleet-type "ON_DEMAND"
--build-id "build-92f061ed-27c9-4a02-b1f4-6f85b2385620"
--certificate-configuration "CertificateType=GENERATED"
--runtime-configuration "GameSessionActivationTimeoutSeconds=300, MaxConcurrentGameSessionActivations=2, ServerProcesses=[{LaunchPath=C:\game\Bin64.dedicated\MultiplayerSampleProjectLauncher_Server.exe, Parameters=+sv_port 33435 +start_lobby, ConcurrentExecutions=10}]",
--new-game-session-protection-policy "FullProtection"
--resource-creation-limit-policy "NewGameSessionsPerCreator=3, PolicyPeriodInMinutes=15"
--ec2-inbound-permissions
"FromPort=33435,ToPort=33435,IpRange=0.0.0.0/0,Protocol=UDP"
"FromPort=33235,ToPort=33235,IpRange=0.0.0.0/0,Protocol=UDP"
--MetricGroups "EMEAfleets"
```

**Copiable version:**

```bash
aws gamelift create-fleet --name "SampleFleet123" --description "The sample test fleet" --ec2-instance-type "c5.large" --fleet-type "ON_DEMAND" --MetricGroups "EMEAfleets" --build-id "build-92f061ed-27c9-4a02-b1f4-6f85b2385620" --certificate-configuration "CertificateType=GENERATED" --runtime-configuration "GameSessionActivationTimeoutSeconds=300, MaxConcurrentGameSessionActivations=2, ServerProcesses=[{LaunchPath=C:\game\Bin64.dedicated\MultiplayerSampleProjectLauncher_Server.exe, Parameters=+sv_port 33435 +start_lobby, ConcurrentExecutions=10}]",
```

**Version:** 107
version "FullProtection" --resource-creation-limit-policy "NewGameSessionsPerCreator=3,PolicyPeriodInMinutes=15" --ec2-inbound-permissions "FromPort=33435,ToPort=33435,IpRange=0.0.0.0/0,Protocol=UDP" "FromPort=33235,ToPort=33235,IpRange=0.0.0.0/0,Protocol=UDP"

If the create-fleet request is successful, Amazon GameLift returns a set of fleet attributes that includes the configuration settings you requested and a new fleet ID. Amazon GameLift immediately initiates the fleet activation process and sets the fleet status to **New**. You can track the fleet's status and view other fleet information using these CLI commands:

- describe-fleet-events
- describe-fleet-attributes
- describe-fleet-capacity
- describe-fleet-port-settings
- describe-fleet-utilization
- describe-runtime-configuration

You can change the fleet's capacity and other configuration settings as needed using these commands:

- update-fleet-attributes
- update-fleet-capacity
- update-fleet-port-settings
- update-runtime-configuration

### Deploy a Realtime Servers Fleet

You can create a new fleet of Realtime game servers to host game sessions for your game. Realtime Servers fleets require that you create a Realtime script and upload it to Amazon GameLift. If you have a custom game server build, see **Deploy a GameLift Fleet for a Custom Game Build (p. 104)** for help creating a fleet with it. Use either the Amazon GameLift console or the AWS Command Line Interface (CLI) to create a fleet. You can change a fleet's configuration by **editing a fleet (p. 111)**.

### Create a Realtime Fleet (Console)

**To create a Realtime fleet with the Amazon GameLift console:**

1. Open the Amazon GameLift console at [https://console.aws.amazon.com/gamelift/](https://console.aws.amazon.com/gamelift/). Go to the **Fleets: Create fleet** page to configure a new fleet.

2. **Fleet Details.**

   - **Name** – Create a meaningful fleet name so you can easily identify it in a list and in metrics.
   - **Description** – (Optional) Add a short description for this fleet to further aid identification.
   - **Fleet type** – Choose whether to use on-demand or spot instances for this fleet. Learn more about fleet types in **Choose Computing Resources (p. 100)**.
   - **Metric group** – (Optional) Enter the name of a new or existing fleet metric group. When using Amazon CloudWatch to track your Amazon GameLift metrics, you can aggregate the metrics for multiple fleets by adding them to the same metric group.
   - **Instance role ARN** – (Optional) Enter the ARN value for an IAM role that you want to associated with this fleet. This setting allows all instances in the fleet to assume the role, which extends access to a defined set of AWS services. Learn more about how to **Access AWS Resources From Your Fleets (p. 45)**.
Deploy a Realtime Servers Fleet

- **Certificate type** – Choose whether to have GameLift generate a TLS certificate for the fleet. With this feature enabled for a Realtime fleet, GameLift automatically authenticates the client/server connection and encrypts all communication between game client and server. Once the fleet is created, you cannot change the certificate type.

- **Binary type** – Select the binary type "Script".

- **Script** – Select the Realtime script you want to deploy from the dropdown list.

3. **Instance type.** Select an Amazon EC2 instance type from the list. The instance types listed vary depending several factors, including the current region, the operating system of the selected game build, and the fleet type (on-demand or spot). Learn more about choosing an instance type in Choose Computing Resources (p. 100). Once the fleet is created, you cannot change the instance type.

4. **Process management.** Configure how you want server processes to run on each instance.
   a. **Server process allocation.**

   Specify the type and number of game server processes you want to run on each instance. Each fleet must have at least one server process configuration defined and can have multiple configurations. For example, if you want to launch processes using different files in your uploaded Realtime script, you must have a configuration for each type of process you want to launch.

   - **Launch path** – Type the name of the script file that you want to launch with. A launch script file must call an `Init()` function. During deployment, your uploaded Realtime script files are unzipped and stored in the `/local/game/` directory, so you just need to specify the script file name. Example: `MyRealtimeLaunchScript.js`.

   - **Launch parameters** – (Optional) You can pass information to your Realtime script at launch time. Type the information as a set of command line parameters. Example: `-map Winter444`.

   - **Concurrent processes** – Indicate how many server processes with this configuration to run concurrently on each instance in the fleet.

   Once you enter a server process configuration, click the green checkmark button on the right to save the configuration. To add additional server process configurations, click **Add configuration**.

   Check the Amazon GameLift limits on the number of concurrent server processes. Limits on concurrent server processes per instance apply to the total of concurrent processes set for all configurations. For example, if you're limited to one process, you can have only one configuration, and concurrent processes must be set to 1. If a fleet is configured to exceed the limit, the fleet cannot activate.

   The collection of server process configurations is called the fleet's runtime configuration. It describes all server processes that will be running on each instance in this fleet at any given time.

   b. **Game session activation (optional):**

   Set the following limits to determine how new game sessions are activated on the instances in this fleet:

   - **Max concurrent game session activation** – Limit the number of game sessions on an instance that can be activating at the same time. This limit is useful when launching multiple new game sessions may have an impact on the performance of other game sessions running on the instance.

   - **New activation timeout** – This setting limits the amount of time Amazon GameLift allows for a new game session to activate. If the game session does not move to status ACTIVE, the game session activation process is terminated.
5. **Protection policy (optional).** Indicate whether or not to apply game session protection to instances in this fleet. Protected instances are not terminated during a scale-down event if they are hosting an active game session. Using this setting applies a fleet-wide protection policy; you can also set protection for individual game sessions when creating the game session.

6. Once you've finished configuring the new fleet, click **Initialize fleet**. Amazon GameLift assigns an ID to the new fleet and begins the fleet activation process. You can view the new fleet's status on the **Fleets** page. Once the fleet is active, you can change the fleet's capacity (p. 122), runtime configuration, and other configuration settings as needed.

### Create a Realtime Fleet (AWS CLI)

To create a Realtime fleet with the AWS CLI, open a command line window and use the `create-fleet` command to define a new fleet. See complete documentation on this command in the [AWS CLI Command Reference](#). Get and install the AWS Command Line Interface tool.

The example `create-fleet` request shown below creates a new fleet with the following characteristics:

- The fleet will use `c5.large` spot instances.
- It will deploy the specified Realtime script.
- Each instance in the fleet will run ten identical processes of the Realtime script concurrently, enabling each instance to host up to ten game sessions simultaneously.
- On each instance, Amazon GameLift will allow only two new game sessions to be activating at the same time. It will also terminate any activating game session if it is not ready to host players within 60 seconds.
- All game sessions hosted on instances in this fleet have game session protection turned on. It can be turned off for individual game sessions.
- Individual players can create three new game sessions within a 15-minute period.
- Metrics for this fleet will be added to the `EMEAfleets` metric group, which (in this example) combines metrics for all fleets in EMEA regions.

**Note**

For Realtime Servers fleets, Amazon GameLift automatically sets TCP and UDP ranges for use by the Realtime servers. You can view the automatic settings by calling the CLI command `describe-fleet-port-settings`.

```
$ aws gamelift create-fleet
  --name "SampleRealtimeFleet123"
  --description "A sample Realtime fleet"
  --ec2-instance-type "c5.large"
  --fleet-type "SPOT"
  --script-id "script-111aaaaa-22bb-33cc-44dd-5555eeee66ff"
  --certificate-configuration "CertificateType=GENERATED"
  --runtime-configuration "GameSessionActivationTimeoutSeconds=60,
                           MaxConcurrentGameSessionActivations=2,
                           ServerProcesses=([{LaunchPath=/local/game/
                              myRealtimeLaunchScript.js,
                              Parameters=+map Winter444,
                              ConcurrentExecutions=10}])"
  --new-game-session-protection-policy "FullProtection"
  --resource-creation-limit-policy "NewGameSessionsPerCreator=3,
                                 PolicyPeriodInMinutes=15"
  --MetricGroups "EMEAfleets"
```

*Copiable version:*

Version 110
aws gamelift create-fleet --name "SampleRealtimeFleet123" --description "A sample Realtime fleet" --ec2-instance-type "c5.large" --fleet-type "SPOT" --script-id "script-1111aaaa-22bb-33cc-44dd-5555eeee66ff" --certificate-configuration "CertificateType=GENERATED" --runtime-configuration "GameSessionActivationTimeoutSeconds=60,MaxConcurrentGameSessionActivations=2,ServerProcesses=[{LaunchPath=/local/game/myRealtimeLaunchScript.js,Parameters=+map Winter444,ConcurrentExecutions=10}]" --new-game-session-protection-policy "FullProtection" --resource-creation-limit-policy "NewGameSessionsPerCreator=3,PolicyPeriodInMinutes=15" --MetricGroups "EMEAfleets"

If the create-fleet request is successful, Amazon GameLift returns a set of fleet attributes that includes the configuration settings you requested and a new fleet ID. Amazon GameLift immediately initiates the fleet activation process and sets the fleet status to **New**. You can track the fleet's status and view other fleet information using these CLI commands:

- describe-fleet-events
- describe-fleet-attributes
- describe-fleet-capacity
- describe-fleet-port-settings
- describe-fleet-utilization
- describe-runtime-configuration

Once the fleet is active, you can change the fleet's capacity and other configuration settings as needed using these commands:

- update-fleet-attributes
- update-fleet-capacity
- update-fleet-port-settings
- update-runtime-configuration

Manage Fleet Records

Use the Amazon GameLift console or the AWS CLI to manage your existing fleets, including updating the fleet's attributes, port settings, and runtime configuration. You can also delete a fleet.

Update a Fleet

Use the **Edit fleet** page in the Amazon GameLift console to change a fleet's configuration. All fleet properties can be changed except for the build ID and the instance type. To change scaling settings, see Auto-Scale Fleet Capacity (p. 123).

**Note**

An active fleet may be deployed with a build that has been deleted or is in an error state. This does not affect the fleet's status or ability to host game sessions. In this situation, you may see a Build status of **Deleted** or **Error** (if an error occurred while retrieving the build info).

GameLift Console

**To update a fleet configuration**

1. Open the Amazon GameLift console at [https://console.aws.amazon.com/gamelift/](https://console.aws.amazon.com/gamelift/).
2. Choose **Fleets** from the menu bar to view a list of fleets, and click on the name of the fleet you want to update. A fleet must be in **ACTIVE** status before it can be edited.
3. On the Fleet detail page, under **Actions**, choose **Edit fleet**.
4. On the Edit fleet page, you can make the following updates (see Deploy a GameLift Fleet for a Custom Game Build (p. 104) for more detailed field descriptions):

- Change the fleet attributes such as Name and Description.
- Add or remove Metric groups, which are used in Amazon CloudWatch to track aggregated Amazon GameLift metrics for multiple fleets.
- Change how you want server processes to run and host game sessions by updating the Server process allocation (runtime configuration) and game session activation settings.
- Update the EC2 port settings used to connect to server processes on this fleet.
- Update resource creation limit settings.
- Turn game session protection on or off.

5. Click Submit to save your changes.

AWS CLI

Use the following AWS CLI commands to update a fleet:

- update-fleet-attributes
- update-fleet-port-settings
- update-runtime-configuration

Delete a Fleet

You can delete a fleet when it is no longer needed. Deleting a fleet permanently removes all data associated with game sessions and player sessions, as well as collected metric data. As an alternative, you can retain the fleet, disable auto-scaling, and manually scale the fleet to 0 instances.

**Note**
If the fleet being deleted has a VPC peering connection, you first need to request authorization by calling CreateVpcPeeringAuthorization. You do not need to explicitly delete the VPC peering connection—this is done as part of the delete fleet process.

You can use either the Amazon GameLift console or the AWS CLI tool to delete a fleet.

GameLift Console

**To delete a fleet**

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. Choose Fleets from the menu bar to view a list of fleets, and click on the name of the fleet you want to delete. Only fleets in ACTIVE or ERROR status can be deleted.
3. On the Fleet detail page for your selected fleet, verify that the fleet has zero active instances. If the fleet still has instances, go to the Scaling tab and do the following:
   - Check the box Disable all scaling policies for the fleet. This action stops all auto-scaling, which would counteract your manual scaling settings.
   - Manually adjust the desired instance count to "0".

   It may take several minutes for the fleet to scale down. If any instances have active game sessions with game session protection, you'll need to either wait for the game sessions to end, or stop protection for the active game sessions (this can't be done in the Console, see UpdateGameSession).
4. Once the fleet is scaled down to zero active instances, you can delete the fleet. At the top of the Fleet detail page, under Actions, choose Terminate fleet.
5. In the **Terminate fleet** dialog box, confirm the deletion by typing the name of the fleet.
6. Click **Delete**.

**AWS CLI**

**To delete a fleet**

Get and install the AWS Command Line Interface tool.

1. In a command line window, call **describe-fleet-capacity** and verify that the fleet to be deleted has been scaled down to zero active instances. If the fleet still has active instances:
   a. call **stop-fleet-actions** to disable auto-scaling.
   b. Call **update-fleet-capacity** and set the parameter **desired-instances** to "0".
   c. Wait for the fleet to scale down to zero active instances. This may take several minutes. If any instances have active game sessions with game session protection, you'll need to either wait for the game sessions to end, or stop protection for the active game sessions (see **update-game-session**).
2. Once the fleet is scaled down, call **delete-fleet** to delete the fleet.

**Add an Alias to a Amazon GameLift Fleet**

An Amazon GameLift **alias** is used to abstract a fleet designation. Fleet designations tell Amazon GameLift where to search for available resources when creating new game sessions for players. By using aliases instead of specific fleet IDs, you can more easily and seamlessly switch player traffic from one fleet to another by changing the alias's target location.

There are two types of routing strategies for aliases:

- **Simple** – A simple alias routes player traffic to a specified fleet ID. You can update the fleet ID for an alias at any time.
- **Terminal** – A terminal alias does not resolve to a fleet. Instead, it passes a message back to the client. For example, you may want to direct players who are using an out-of-date client to a location where they can get an upgrade.

Fleets have a finite lifespan, and there are several reasons why you'll need to switch out fleets during the life of a game. Specifically, you can't update a fleet's game server build or change certain computing resource attributes (instance types, spot/on-demand usage) on an existing fleet. Instead, you need to create new fleets with the changes and then switch players to the new fleets. With aliases, switching fleets has minimal impact on your game and is invisible to players.

Aliases are primarily useful in games that do not use queues. Switching fleets in a queue is a simple matter of creating a new fleet, adding it to the queue, and removing the old fleet, none of which is visible to players. In contrast, game clients that don't use queues must specify which fleet to use when communicating with the Amazon GameLift service. Without aliases, a fleet switch requires updates to your game code and possibly the need to distribute updated game clients to players. With aliases, you can avoid both.

**Create a New Alias**

You can create a new alias to resolve to a fleet.

**To create a new alias**

1. Open the Amazon GameLift console at [https://console.aws.amazon.com/gamelift/](https://console.aws.amazon.com/gamelift/).
2. Choose Aliases from the menu bar.
3. On the Aliases page, click Create alias.
4. On the Create alias page, in the Alias details section, do the following:
   • Alias name – Type a friendly name so you can easily identify the alias in the catalog.
   • Description – (Optional) Type a short description for your alias to add further identification.
5. In the Routing options section, for Type, choose Simple or Terminal:
   • If you choose Simple, select an available fleet to associate with your alias. A simple alias routes player traffic to the associated fleet.
   • If you select Terminal, type a message that will be displayed to players. A terminal alias does not resolve to a fleet but only passes your message to the client.
6. Click Configure alias.

Edit an Alias

Use the Edit alias page in the Amazon GameLift console to update the information for your alias.

To edit an alias
1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. Choose Aliases from the menu bar.
3. On the Aliases page, click the name of the alias you want to edit.
4. On the selected alias page, for Actions, choose Edit alias.
5. On the Edit alias page, you can edit the following:
   • Alias name – Friendly name for your alias.
   • Description – Short description for your alias.
   • Type – Routing strategy for player traffic. Select Simple to change the associated fleet or select Terminal to edit the termination message.
6. Click Submit.

Debug Fleet Issues

This topic provides guidance on the fleet-related issues. For additional troubleshooting, you can remotely access a fleet instance. See Remotely Access Fleet Instances (p. 117).

Fleet Creation Issues

How Fleet Creation Works

When you create a new fleet, the Amazon GameLift service performs a series of tasks to prepare a single instance based on the fleet's configuration. As each phase of fleet creation is completed, a series of events are emitted for the fleet along with the fleet's current status. You can track all events, including those for fleet creation, using the Amazon GameLift console (see the Fleet detail page, Events tab.

Problems during fleet creation will cause the fleet status to go to Error status with meaningful error messaging. The phase in which fleet creation failed can also be a good indicator. Fleet creation phases are:
   • New – Fleet record is created. Resources are allocated for the initial instance.
• **Downloading** – The game build files are downloaded to the instance and extracted.
• **Validating** – The downloaded game build files are validated.
• **Building** – The game server build is installed on the instance, using an install script if available.
• **Activating** – Based on the fleet runtime configuration, server process(es) are started. At least one process must communicate with Amazon GameLift to report its readiness to host a game session.

As soon as one server process notifies Amazon GameLift that it is ready, the fleet status moves to **Active**.

### Common Problems

#### Downloading and validating

During this phase, fleet creation may fail if there are issues with the extracted build files, the installation script won't run, or if the executable(s) designated in the runtime configuration is not included in the build files. Amazon GameLift provides logs related to each of these issues.

If the logs do not reveal an issue, it's possible that the problem is due to an internal service error. In this case, try to create the fleet again. If the problem persists, consider re-uploading the game build (in case the files were corrupted). You can also contact Amazon GameLift support or post a question on the forum.

#### Building

Issues that cause failure during the build phase are almost certainly due to problems with the game build files and/or the installation script. Verify that your game build files, as uploaded to Amazon GameLift, can be installed on a machine running the appropriate operating system. Be sure to use a clean OS installation, not an existing development environment.

#### Activating

The most common fleet creation problems occur during the **Activating** phase. During this phase, a number of elements are being tested, including the game server's viability, the runtime configuration settings, and the game server's ability to interact with the Amazon GameLift service using the Server SDK. Common issues that come up during fleet activation include:

**Server processes fail to start.**

First check that you've correctly set the launch path and optional launch parameters in the fleet's runtime configuration. You can view the fleet's current runtime configuration using either the Amazon GameLift console (see the **Fleet** detail page, **Capacity allocation** (p. 175)) tab or by calling the AWS CLI command `describe-runtime-configuration`. If the runtime configuration looks correct, check for issues with your game build files and/or installation script.

**Server processes start but fleet fails to activate.**

If server processes start and run successfully, but the fleet does not move to **Active** status, a likely cause is that the server process is failing to notify Amazon GameLift that it is ready to host game sessions. Check that your game server is correctly calling the Server API action `ProcessReady()` (see **Prepare a Server Process** (p. 42)).

**VPC peering connection request failed.**

For fleets that are created with a VPC peering connection (see **Create a New Fleet with VPC Peering** (p. 167)), VPC peering is done during this **Activating** phases. If a VPC peering fails for any reason, the new fleet will fail to move to **Active** status. You can track the success or failure of the peering request by calling `describe-vpc-peering-connections`. Be sure to check that a valid VPC peering authorization exists (`describe-vpc-peering-authorizations`, since authorizations are only valid for 24 hours).
Server Process Issues

Server processes start but fail quickly or report poor health.

Other than issues with your game build, this outcome can happen when trying to run too many server processes simultaneously on the instance. The optimum number of concurrent processes depends on both the instance type and your game server's resource requirements. Try reducing the number of concurrent processes, which is set in the fleet's runtime configuration, to see if performance improves. You can change a fleet's runtime configuration using either the Amazon GameLift console (edit the fleet's capacity allocation settings) or by calling the AWS CLI command update-runtime-configuration.

Fleet Deletion Issues

Fleet can't be terminated due to max instance count.

The error message indicates that the fleet being deleted still has active instances, which is not allowed. You must first scale a fleet down to zero active instances. This is done by manually setting the fleet's desired instance count to "0" and then waiting for the scale-down to take effect. Be sure to turn off auto-scaling, which will counteract manual settings.

VPC actions are not authorized.

This issue only applies to fleets that you have specifically created VPC peering connections for (see VPC Peering for Amazon GameLift (p. 165)). This scenario occurs because the process of deleting a fleet also includes deleting the fleet's VPC and any VPC peering connections. You must first get an authorization by calling the GameLift service API CreateVpcPeeringAuthorization() or use the AWS CLI command create-vpc-peering-authorization. Once you have the authorization, you can delete the fleet.

Realtime Servers Fleet Issues

Zombie game sessions: They start and run a game, but they never end.

You might observe this issues as any of the following scenarios:

- Script updates are not picked up by the fleet's Realtime servers.
- The fleet quickly reaches maximum capacity and does not scale down when player activity (such as new game session requests) decreases.

This is almost certainly a result of failing to successfully call processEnding in your Realtime script. Although the fleet goes active and game sessions are started, there is no method for stopping them. As a result, the Realtime server that is running the game session is never freed up to start a new one, and new game sessions can only start when new Realtime servers are spun up. In addition, updates to the Realtime script do not impact already-running game sessions, only ones.

To prevent this from happening, scripts need to provide a mechanism to trigger a processEnding call. As illustrated in the Realtime Servers Script Example (p. 66), one way is to program an idle session timeout where, if no player is connected for a certain amount of time, the script will end the current game session.

However, if you do fall into this scenario, there are a couple workarounds to get your Realtime servers unstuck. The trick is to trigger the Realtime server processes—or the underlying fleet instances—to restart. In this event, GameLift automatically closes the game sessions for you. Once Realtime servers are freed up, they can start new game sessions using the latest version of the Realtime script.

There are a couple of methods to achieve this, depending on how pervasive the problem is:
• Scale the entire fleet down. This method is the simplest to do but has a widespread effect. Scale the fleet down to zero instances, wait for the fleet to fully scale down, and then scale it back up. This will wipe out all existing game sessions, and let you start fresh with the most recently updated Realtime script.

• Remotely access the instance and restart the process. This is a good option if you have only a few processes to fix. If you are already logged onto the instance, such as to tail logs or debug, then this may be the quickest method. See Remotely Access Fleet Instances (p. 117).

If you opt not to include way to call `processEnding` in your Realtime script, there are a couple of tricky situations that might occur even when the fleet goes active and game sessions are started. First, a running game session does not end. As a result, the server process that is running that game session is never free to start a new game session. Second, the Realtime server does not pick up any script updates.

## Remotely Access Fleet Instances

You can remotely access any fleet instance that is currently running in an Amazon GameLift fleet. This capability is useful for troubleshooting fleet activation issues. You can also use this feature to get real-time game server activity, such as track log updates or run benchmarking tools using actual player traffic.

When remotely accessing individual Amazon GameLift instances, keep the following in mind:

• The Amazon GameLift service continues to manage fleet activity and capacity. Establishing a remote connection to an instance does not affect how Amazon GameLift manages it in any way. As a result, the instance continues to execute the fleet runtime configuration, stop and start server processes, create and terminate game sessions, and allow player connections. In addition, the Amazon GameLift service may terminate the instance at any time as part of a scale down event.

• Making local changes to an instance that is hosting active game sessions and has live players connected may significantly affect player experience. For example, your local changes have the potential to drop individual players, crash game sessions or even shut down the entire instance with multiple game sessions and players affected.

For more information on how games are deployed and managed on Amazon GameLift instances, see the following topics:

• How Amazon GameLift Works (p. 3)
• Debug Fleet Issues (p. 114)
• How a Fleet Manages Multiple Processes (p. 102)

## Connect to an Instance

You can access remote instances that are running either Windows or Linux. To connect to a Windows instance, use a remote desktop protocol (RDP) client. To connect to a Linux instance, use an SSH client.

Use the AWS CLI get the information you need to access a remote instance. For help, see the AWS CLI Command Reference. Get and install the AWS Command Line Interface tool. You can also use the AWS SDK, with documentation available in the Amazon GameLift Service API Reference.

1. **Find the ID of the instance you want to connect to.** When requesting access, you must specify an instance ID. Use the AWS CLI command `describe-instances` (or the API call `DescribeInstances`) with a fleet ID to get information on all instances in the fleet. For help, including example requests and responses, see the CLI or API reference guides.

2. **Request access credentials for the instance.** Once you have an instance ID, use the command `get-instance-access` (or the API call `GetInstanceAccess`) to request access credentials and other
information. For help, including example requests and responses, see the CLI or API reference guides. If successful, Amazon GameLift returns the instance's operating system, IP address, and a set of credentials (user name and secret key). The credentials format depends on the instance operating system. Use the following instructions to retrieve credentials for either RDP or SSH.

- **For Windows instances** – To connect to a Windows instance, RDP requires a user name and password. The `get-instance-access` request returns these values as simple strings, so you can use the returned values as is. Example credentials:

  ```json
  "Credentials": {
    "Secret": "aA1bBB2CcCd3EEd",  
    "UserName": "gl-user-remote"
  }
  ```

- **For Linux instances** – To connect to a Linux instance, SSH requires a user name and private key. Amazon GameLift issues RSA private keys and returns them as a single string, with the newline character (`\n`) indicating line breaks. To make the private key usable, you must (1) convert the string to a `.pem` file, and (2) set permissions for the new file. Example credentials returned:

  ```json
  "Credentials": {
    "Secret": "-----BEGIN RSA PRIVATE KEY-----
    nEXAMFLbZiCAQEMQW7w2aDsrA1M3nR1qtvWyGRX8mygDaR7/gx42kWXs74vXe/bbCpgSgje/
    /vnB0u7JLxix9PsNnpnB+Dc21eyz6Cv+jyh7mXaJwJ6JWS/A/kH1750s5rCt7bQwR2w5VpU6dEGWQ
    /n/ahxXmNjQx6XQg/f1NvXVbwrerqQ+2WqegIu6whMKzrELeJFLHIxTYToQzUGmJUm9ove449lLx91f
    /iG50TCEf0sl8ddgqCPy6bPa1i1ju9Xx/aZorByV+tpU0zEL+wmxnt5t/nHq5xvD9JH7km6SuPW
    /noPzev/DBV+x+bbhtShsFr9Y7VdfQjfbWwHx1gBtc2cU2/We18b/HYWldmQAAbAoBAG1z1aEvnruq
    /n/ulr7Vln5m7NLSW4hJLAIW6tUT/fzvttcKH0skbCQXuriHjvM2MQy7/X0k2NfjLV/ufGx/b1l
    nmbq5wMGUinEpjA3D6QSSs3kICLwWYUJiGc0usiSbhJaoj/  
    GTL0U5W5Mc6V3pA1BvHZ2bH2b\n    bahyWy7NjFje4M8b6y2Y13V2cmX+X/  
    BOsShnJ36+hjXPFWm3NZeNmcjJa/K15Ymhj/  
    tJSW9D/\n     /nI0gK9TopEp7CKfIfatEAyYyZ1vQogRg6k64iuM9jkaA30axZMqexVXJ1LZVHEB7hIy9d8O5zR  
    /nq0s/FiZgXa2iiJcWv0v1ple73+kCqyEAXwtyhKxIFwpwSN1Af0L8oAbhjyBy72S5Qgfl  
    +liP1nYrXkLm0bLX1vRAh+yHPI021hNOJ7n3A4xv/cpg09gbU31+43eY3247G/UnH/GfbjxsOxQx/  
    /n/+P9otyQcF7sQ5T5A5P2v+wk5k0BEKzet9XcKwOBtYLGhnePev7CgYEA06VgovaY6HeHui9Kluw  
    /nayav0e1c5zkjF9mHfJFRy21iritw2vdpn/9g4lUIrrZpWOVhivm=xtma2Zs1p/ltk75xDwun  
    /n/WA8gKn80M3Q9feqy74sTiQvU8AAdf1f5bR1lIvGvTeT0HLYXpJnEKHu+unI2aljWivUt5spBrBoUC  
    /n/gYbjbo+02zksOcCp2292bziy1pIdcDbrsyfXyRX5v2uNqwA/jldp9Fnp295yQ+bXMBXlYcyWQv0wBh  
    /n/on7owykaBY7OzdswqewBqQs2v3wDMx4nGdESIfXw15sKuaAeOCObosy18w8fxc0j5t12sdoxNeGd  
    /n/Arq6Gw/G16sQUA29zK9vwwKBeF/09vI/1wJbirsDQzvwHFEPPrTkJvJYX72Lt69ezxlsjgFKshy  
    /n/W8b4xH2mzCmpCqBPlAYm5rj/\n    /TolbxyArOIM10W1ANXMG4BKSy1mzSvAQo+q+rCj3ddoyPlj  
    /n/\n    jmbjOed/nyF1rldNxAyVhH9BbSdakZ2fIE6bEREK5R9smRaAGAMTwYaez7vAfK/8Sfy0u1OeQgDa  
    /n/0UWHU38vvdNDcgBIPXDSH3nG3ABcjuj1jmwb1iwvBypjy7u0GdWyWtnO1xtdn6nQ2a11rnP3qG  
    /n\n    /vY8KkAKKfY6j3kuUFKtw2cXWFYJXkfcfrR/fQL509ndHAKJYX74e4j5LncTz3sPzywAc\n    /n----END RSA PRIVATE KEY------",  
    "UserName": "gl-user-remote"
  }
  ```

When using the AWS CLI, you can automatically generate a properly formatted `.pem` file by including the `--query` and `--output` parameters to your `get-instance-access` command.

To set permissions on the `.pem` file, run the following command:

```
chmod 400 MyPrivateKey.pem
```

3. **Open a port for the remote connection.** Instances in Amazon GameLift fleets can only be accessed through ports authorized in the fleet configuration. You can view a fleet's port settings using the command `describe-flight-port-settings`.

As a best practice, we recommend opening ports for remote access only when you need them and closing them when you are finished. Use the command `update-flight-port-settings` to add a port setting for the remote connection (such as 22 for SSH or 3389 for RDP). For the IP range
value, specify the IP addresses for the devices you plan to use to connect (converted to CIDR format).
Example:

```bash
$ aws gamelift update-fleet-port-settings
  --fleet-id "fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa"
  --inbound-permission-authorizations
    "FromPort=22,ToPort=22,IpRange=54.186.139.221/32,Protocol=TCP"
```

4. **Open a remote connection client.** Use Remote Desktop for Windows or SSH for Linux instances. Connect to the instance using the IP address, port setting, and access credentials.

   SSH example:

   ```
   ssh -i MyPrivateKey.pem gl-user-remote@192.0.2.0
   ```

**View and Update Remote Instances**

When connected to an instance remotely, you have full user and administrative access. This means you also have the ability to cause errors and failures in game hosting. If the instance is hosting games with active players, you run the risk of crashing game sessions and dropping players, as well as disrupting game shutdown processes and causing errors in saved game data and logs.

Hosting resources on an instance can be found in the following locations:

- **Game build files.** These are the files included in the game build you uploaded to Amazon GameLift. They include one or more game server executables, assets and dependencies. These files are located in a root directory called `game`:
  - On Windows: `c:\game`
  - On Linux: `/local/game`

- **Game log files.** Any log files your game server generates are stored in the `game` root directory at whatever directory path you designated.

- **Amazon GameLift hosting resources.** Files used by the Amazon GameLift service to manage game hosting are located in a root directory called `Whitewater`. These files should not be changed for any reason.

- **Runtime configuration.** The fleet runtime configuration is not accessible for individual instances. To test changes to a runtime configuration (launch path, launch parameters, maximum number of concurrent processes), you must update the fleet-wide runtime configuration (see the AWS SDK action `UpdateRuntimeConfiguration` or the AWS CLI `update-runtime-configuration`).

- **TLS certificates.** If the instance is on a fleet that has TLS certificate generation enabled, certificate files, including the certificate, certificate chain, private key, and root certificate, are stored in the following location:
  - On Windows: `c:\\GameMetadata\\Certificates`
  - On Linux: `/local/gamemetadata/certificates/`

**Scaling Amazon GameLift Fleet Capacity**

Fleet capacity, measured in instances, determines the number of game sessions (and players) the fleet can host. One of the most challenging tasks with game hosting is to strike a balance between maintaining enough capacity to accommodate every new player and not wasting money on unnecessary resources. Learn more about how capacity scaling works (p. 6) in Amazon GameLift.

You have full control over scaling a fleet. You can set capacity to a specific number of instances, or you can take advantage of auto-scaling to adjust capacity based on actual player demand. We recommended...
that you start by turning on the auto-scaling option Target Tracking. Target tracking is an effective and easy to use tool for that helps you maintain just enough hosting resource to accommodate current player demand and handle sudden spikes. For the vast majority games, target tracking will be the only solution you need.

The topics in this section provide detailed help with the following tasks:

- Set minimum and maximum limits for capacity scaling (p. 120)
- Manually set capacity levels (p. 122)
- Turn on auto-scaling with Target Tracking (p. 124)
- Manage rule-based auto-scaling (advanced feature) (p. 125)
- Temporarily disable auto-scaling (p. 122)

Most fleet scaling activities can be done using the Amazon GameLift console. You can also use the AWS SDK or AWS CLI with the Amazon GameLift Service API for all fleet scaling.

**To access fleet capacity settings in the console:**

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the **Fleets** page, click the name of an active fleet to open the fleet's detail page. (You can also access a fleet's detail page via the **Dashboard**.)
3. Open the **Scaling** tab to view historical scaling metrics and to view or change the current settings. Settings are located below the metrics graph. In this section, you can view or update scaling limits, manually set fleet capacity, enable or disable auto-scaling, turn on target-based auto-scaling, and view all active auto-scaling policies.

**Topics**

- Set Fleet Capacity Limits (p. 120)
- Manually Set Fleet Capacity (p. 122)
- Auto-Scale Fleet Capacity (p. 123)

### Set Fleet Capacity Limits

Fleet size is determined by the number of instances it contains. Each fleet has a defined minimum and maximum limit, which you can set as needed for your game. All requests to change fleet capacity (either by auto-scaling or manual adjustment) must fall within the current limits. By default, limits on new fleets are set to a minimum of 0 instances and a maximum of 1 instance. Before scaling up a fleet, you'll need to adjust the fleet's limits.

If you're auto-scaling a fleet, the maximum limit lets Amazon GameLift scale up the fleet as needed to accommodate player demand while also preventing runaway hosting costs, such as might occur during a DDOS attack. Set up CloudWatch to alarm when capacity approaches the maximum limit, so you can evaluate the situation and manually adjust as needed. (You can also set up a billing alert to monitor AWS costs.) The minimum limit is useful when you want to ensure that you always have some hosting availability at all times.

Limits also apply to manually scaled fleets. Before you can adjust fleet capacity to a value outside the limit range, you must change the limits. Since fleet capacity has such a significant effect on game availability and player experience, the limits feature provides an additional layer of control over capacity.

You can set a fleet's capacity limits in the Amazon GameLift console or by using the AWS CLI. The fleet's status must be **Active**.
To Set Capacity Limits (Console)

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the Fleets page, click the name of an active fleet to open the fleet's detail page. (You can also access a fleet's detail page via the Dashboard.)
3. Open the Scaling tab to view historical scaling metrics and to view or change the current settings. Scaling settings are located below the metrics graph.
4. Under Instance Limits, set minimum and maximum instance counts. For each control, commit your changes by clicking the checkmark button.

If the fleet's current desired instance count is outside the new limit range, you'll get an error. In this case, you must first adjust the fleet's desired instance count so that it falls inside the new limit range. This can be done on the Scaling tab. If the fleet uses auto-scaling, you'll need to disable auto-scaling, manually adjust the desired instance count and set the new limit range, and then re-enable auto-scaling.

The new limits are immediately reflected in the graph at the top of the Scaling tab.

To Set Capacity Limits (AWS CLI)

1. Check current capacity settings. In a command line window, use the describe-fleet-capacity command with the fleet ID of the fleet you want to change capacity for. This command returns a FleetCapacity object that includes the current instance count and capacity limits. Determine whether the new instance limits will accommodate the current desired instances setting.

   ```
   aws gamelift describe-fleet-capacity --fleet-id <unique fleet identifier>
   ```

2. Update limit settings. In a command line window, use the update-fleet-capacity command with the following parameters. You can adjust both instance limits and desired instance count with the same command.

   ```
   --fleet-id <unique fleet identifier>
   --max-size <maximum capacity for auto-scaling>
   --min-size <minimum capacity for auto-scaling>
   --desired-instances <fleet capacity as an integer>  [Optional]
   ```

   Example:

   ```
   aws gamelift update-fleet-capacity
   --fleet-id fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa
   --max-size 10
   --min-size 1
   --desired-instances 10
   ```

   Copyable version:

   ```
   aws gamelift update-fleet-capacity --fleet-id
   fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa --max-size 10 --min-size 1 --desired-instances 10
   ```

   If your request is successful, the fleet ID is returned. If the new max-size or min-size value conflicts with the current desired-instances setting, an error is returned.
Manually Set Fleet Capacity

When you create a new fleet, fleet capacity is automatically set to one instance. In response, Amazon GameLift starts one new instance with game server processes as configured. To change fleet capacity, you can either turn on auto-scaling or you can manually set the number of instances you want for the fleet. Learn more about Scaling Fleet Capacity (p. 6).

Setting a fleet's capacity manually can be useful when auto-scaling is not needed or when you need to hold capacity at an arbitrary number of instances temporarily or permanently. When manually setting a desired capacity, keep in mind that this action will affect actual fleet capacity only if (1) there are no auto-scaling policies for the fleet, or (2) auto-scaling has been disabled. If auto-scaling is enabled, it will immediately reset the desired capacity based on its own scaling rules.

You can manually set fleet capacity in the Amazon GameLift console or by using the AWS CLI. The fleet's status must be Active.

Disabling Auto-Scaling

Disabling auto-scaling allows you to turn off all auto-scaling for a fleet and return to manual scaling. While you can always delete a fleet's autoscaling policy, this feature lets you temporarily turn off auto-scaling but retain the policies for future use. For example, if you want to scale up in preparation for an upcoming major event, you can disable auto-scaling, manually set a desired fleet capacity, and then, once the event is in progress, re-enable auto-scaling. When auto-scaling is disabled, all auto-scaling activities for the fleet are stopped, including all currently active policies and any policies that might be created in the future. You can enable/disable auto-scaling in the Amazon GameLift console (see Step 4 in "To manually set capacity").

To Manually Set Capacity (Console)

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the Fleets page, click the name of an active fleet to open the fleet's detail page. (You can also access a fleet's detail page via the Dashboard.)
3. Open the Scaling tab to view historical scaling metrics and to view or change the current settings. Scaling settings are located below the metrics graph.
4. Under Auto-Scaling Policies, check the box "Disable all scaling policies" and commit your change by clicking the check mark button . This setting stops all auto-scaling actions for the fleet. It is a good idea to do this even if the fleet currently has no policies, as it will prevent any newly created policies from taking effect. Once you commit this change, the table listing active scaling policies indicates that all policies are "disabled".
5. Under Auto-Scaling Policies, for the option "Manually adjust the desired instance count to...", specify the number of instances for the fleet. This value tells Amazon GameLift how many instances to maintain in an active state, ready to host game sessions. Commit the change by clicking the checkmark button .

If the new desired instances value is outside the fleet's capacity limits, you'll get an error. In this case, you must first adjust the fleet's instance limits to allow for the new desired instance count. Instance Limits can also be set on the Scaling tab.

As soon as you commit changes to instance limits and manual scaling levels, the new values are reflected in the graph at the top of the Scaling tab. Amazon GameLift immediately begins responding to the changes by deploying additional instances or shutting down unneeded ones. As this process is completed, the number of Active instances change to match the newly updated desired value. This process may take a little time.
To Manually Set Capacity (AWS CLI)

1. **Check current capacity settings.** In a command line window, use the `describe-fleet-capacity` command with the fleet ID of the fleet you want to change capacity for. This command returns a `FleetCapacity` object that includes the current instance count and capacity limits. Determine whether the new instance count falls between the minimum and maximum limits.

   ```bash
   aws gamelift describe-fleet-capacity --fleet-id <unique fleet identifier>
   ```

2. **Update desired capacity.** Use the `update-fleet-capacity` command with the fleet ID and a new `desired-instances` value. If this value falls outside the current limit range, include adjust limit values in the same command.

   ```bash
   --fleet-id <unique fleet identifier>
   --desired-instances <fleet capacity as an integer>
   --max-size <maximum capacity for auto-scaling> [Optional]
   --min-size <minimum capacity for auto-scaling> [Optional]
   ```

   **Example:**

   ```bash
   aws gamelift update-fleet-capacity
   --fleet-id fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa
   --desired-instances 5
   --max-size 10
   --min-size 1
   ```

   **Copyable version:**

   ```bash
   aws gamelift update-fleet-capacity --fleet-id
   fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa --desired-instances 5 --max-size 10 --min-size 1
   ```

   If your request is successful, the fleet ID is returned. If the new `desired-instances` setting is outside the minimum/maximum limits, an error is returned.

## Auto-Scale Fleet Capacity

Use auto-scaling to dynamically scale your fleet capacity in response to game server activity. As players arrive and start game sessions, auto-scaling can add more instances; as player demand wanes, auto-scaling can terminate unneeded instances. Auto-scaling is an effective way to minimize your hosting resources and costs, while still providing a smooth and fast player experience. Learn more about how auto-scaling (p. 6) works in Amazon GameLift.

Auto-scaling is done by creating scaling policies that provide instructions to Amazon GameLift for scaling up or down. There are two types of scaling policies, target-based and rule-based. The target-based approach—target tracking—offers a complete solution; it is recommended as the simplest and most effective option. Rule-based scaling policies, which require you to define each aspect of the auto-scaling decision-making process, is useful for addressing specific issues. It works best as a supplement to target-based auto-scaling.

Target-based auto-scaling can be managed using either the Amazon GameLift Console or the AWS CLI or AWS SDK. Rule-based auto-scaling is managed using the AWS CLI or AWS SDK only, although you can view rule-based scaling policies in the Console.

**Topics**
Auto-Scale with Target Tracking

Target tracking adjusts capacity levels based on a single key fleet metric: "percent available game sessions". This metric measures the number of available game session slots at current capacity—additional game sessions that could be started immediately. In effect, this metric represents the fleet's buffer against sudden increases in player demand.

The primary reason for maintaining a capacity buffer is player wait time. When game session slots are ready and waiting, the time it takes to get new players into game sessions can be measured in seconds. If no resources are available, players must wait for existing game sessions to end or for new resources to become available. The time required to start up new instances and server processes can be minutes.

When setting up target tracking, you simply specify the size of the buffer you want the fleet to maintain. Since the metric "percent available game sessions" measures the percentage of available resources, the actual buffer size is a percentage of the total fleet capacity. Amazon GameLift adds or removes as many instances as are needed to maintain the target buffer size. Choosing a buffer size depends on how you want to prioritize minimizing player wait time against controlling hosting costs. With a large buffer, you minimize wait time but you also pay for extra resources that may not get used. If your players are more tolerant of wait times, you can lower costs by setting a small buffer.

Set Target Tracking (Console)

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the Fleets page, click the name of an active fleet to open the fleet’s detail page. (You can also access a fleet's detail page via the Dashboard.)
3. Open the Scaling tab. This tab displays the fleet’s historical scaling metrics and contains controls for adjusting current scaling settings. Scaling settings are located below the metrics graph.
4. Under Instance Limits, check that the minimum and maximum limits are appropriate for the fleet. With auto-scaling enabled, capacity may adjust to any level between these two limits.
5. Under Auto-Scaling Policies, check the option to Maintain a buffer of X percent game session availability. Set a buffer size, and click the checkmark button to save the auto-scaling settings. Once you’ve saved the settings, a new target-based policy is added to the Scaling policies table.
6. To turn on auto-scaling for the fleet, verify that the option to Disable all scaling policies in the fleet is unchecked. If this option is checked, all policies, including the new target-tracking policy, are disabled. This state is reflected in the Scaling policies table.

Set Target Tracking (AWS CLI)

1. Set capacity limits. Set either or both limit values using the update-fleet-capacity command. For help, see To Set Capacity Limits (AWS CLI) (p. 121).
2. Create a new policy. Open a command-line window and use the put-scaling-policy command with your policy’s parameter settings. To update an existing policy, specify the policy’s name and provide a complete version of the updated policy.

```
--fleet-id <unique fleet identifier>
--name "<unique policy name>"
--policy-type <target- or rule-based policy>
--metric-name <name of metric>
--target-configuration <buffer size>
```
Auto-Scale with Rule-Based Policies

Rule-based scaling policies provide fine-grained control when auto-scaling a fleet's capacity in response to player activity. For each policy, you can link fleet scaling to one of several available fleet metrics, identify a trigger point, and customize the responding scale-up or scale-down event. Rule-based policies are particularly useful for supplementing target-based scaling to handle special circumstances.

A rule-based policy makes the following statement: "If a fleet metric meets or crosses a threshold value for a certain length of time, then change the fleet's capacity by a specified amount." This topic describes the syntax used to construct a policy statement and provides help with creating and managing your rule-based policies.

Manage Rule-Based Policies

Create, update, or delete rule-based policies using the AWS SDK or AWS CLI with the Amazon GameLift Service API. You can view all active policies in the Amazon GameLift console.

To temporarily disable all scaling policies for a fleet, use the AWS CLI command `stop-fleet-actions`.

To create or update a rule-based scaling policy (AWS CLI):

1. **Set capacity limits.** Set either or both limit values using the `update-fleet-capacity` command. For help, see To Set Capacity Limits (AWS CLI) (p. 121).

2. **Create a new policy.** Open a command-line window and use the `put-scaling-policy` command with your policy's parameter settings. To update an existing policy, specify the policy's name and provide a complete version of the updated policy.

    ```bash
    --fleet-id <unique fleet identifier>
    --name "<unique policy name>"
    --policy-type <target- or rule-based policy>
    --metric-name <name of metric>
    --comparison-operator <comparison operator>
    --threshold <threshold integer value>
    --evaluation-periods <number of minutes>
    --scaling-adjustment-type <adjustment type>
    --scaling-adjustment <adjustment amount>
    ```

Example:

```bash
aws gamelift put-scaling-policy
```
Auto-Scale Fleet Capacity

--fleet-id fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa
--name "Scale up when AGS<50"
--policy-type RuleBased
--metric-name AvailableGameSessions
--comparison-operator LessThanThreshold
--threshold 50
--evaluation-periods 10
--scaling-adjustment-type ChangeInCapacity
--scaling-adjustment 1

Copyable version:

aws gamelift put-scaling-policy --fleet-id fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa
--name "Scale up when AGS<50" --policy-type RuleBased --metric-name
AvailableGameSessions --comparison-operator LessThanThreshold --threshold 50 --
evaluation-periods 10 --scaling-adjustment-type ChangeInCapacity --scaling-adjustment 1

To delete a rule-based scaling policy using the AWS CLI:

- Open a command-line window and use the delete-scaling-policy command with the fleet ID and policy name.

Example:

aws gamelift delete-scaling-policy
--fleet-id fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa
--name "Scale up when AGS<50"

Copyable version:

aws gamelift delete-scaling-policy --fleet-id
fleet-2222bbbb-33cc-44dd-55ee-6666ffff77aa --name "Scale up when AGS<50"

Syntax for Auto-Scaling Rules

To construct rule-based scaling policy statement, you must specify six variables:

- If <metric name> remains <comparison operator> <threshold value> for <evaluation period>, then change fleet capacity using <adjustment type> to/by <adjustment value>.

For example, this policy statement triggers a scale-up event whenever a fleet's extra capacity (available hosting resources not currently in use) is less than what is needed to handle 50 new game sessions:

If AvailableGameSessions remains at less than 50 for 10 minutes, then change fleet capacity
using ChangeInCapacity by 1 instances.

Metric name

To trigger a scaling event, link an auto-scaling policy to one of the following fleet-specific metrics. See Amazon GameLift Metrics for Fleets (p. 178) for more complete metric descriptions.

- Activating game sessions
- Active game sessions
- Available game sessions
- Percent available game sessions
Auto-Scale Fleet Capacity

- Active instances
- Available player sessions
- Current player sessions
- Idle instances
- Percent idle instances

The following metrics may be used if the fleet is included in a game session queue:
- Queue depth (fleet specific) – The number of pending game session requests for which this fleet is the best available hosting location.
- Wait time (fleet specific) – Fleet-specific wait time. The length of time that the oldest pending game session request has been waiting to be fulfilled. As with queue depth, this metric reflects only game session requests for which this fleet is the best available hosting location. A fleet's wait time is equal to the oldest current request's time in queue.

Comparison operator

This variable tells Amazon GameLift how to compare the metric data to the threshold value. Valid comparison operators include greater than (＞), less than (＜), greater than or equal (≥), or less than or equal (≤).

Threshold value

When the specified metric value meets or crosses the threshold value, it can trigger a scaling event. Depending on the metric selected, it may indicate an amount of player sessions, game sessions, instances, or game session requests. This value is always a positive integer.

Evaluation period

The metric must meet or cross the threshold value for the full length of the evaluation period before triggering a scaling event. The evaluation period length is consecutive; if the metric retreats from the threshold, the evaluation period starts over again.

Adjustment type and value

This set of variables works together to specify how Amazon GameLift should adjust the fleet's capacity when a scaling event is triggered. Choose from three possible types of adjustments:
- **Change in capacity** – Increase or decrease the current capacity by a specified number of instances. Set the adjustment value to the number of instances to add or remove from the fleet. Positive values add instances, while negative values remove instances. For example, an value of "-10" will scale down the fleet by 10 instances, regardless of the fleet's total size.
- **Percent change in capacity** – Increase or decrease the current capacity by a specified percentage. Set the adjustment value to the percentage you want to increase or decrease the fleet capacity by. Positive values add instances, while negative values remove instances. For example, for a fleet with 50 instances, a percentage change of "20" will add ten instances to the fleet.
- **Exact capacity** – Set desired instances to a specific value. Set the adjustment value to the exact number of instances that you want to maintain in the fleet.

Tips for Rule-Based Auto-Scaling

The following suggestions can help you get the most out of auto-scaling with rule-based policies.

**Use multiple policies**

You can have multiple auto-scaling policies in force for a fleet at the same time. The most common scenario is to have a target-based policy manage most scaling needs and use rule-based policies to handle edge cases. However, there are no limits on using multiple policies.
Multiple policies behave independently. Keep in mind that there is no way to control the sequence of scaling events. For example, if you have multiple policies driving scaling up, it is possible that player activity could trigger multiple scaling events simultaneously. For example, the effects of two scale up policies can easily be compounded if it is possible for player activity to trigger both metrics. Also watch for policies that trigger each other. For example, you could create an infinite loop if you create scale up and scale down policies that sets capacity beyond the threshold of each other.

**Set maximum and minimum capacity**

Each fleet has a maximum and minimum capacity limit. This feature is particularly important when using auto-scaling. Auto-scaling will never set capacity to a value outside of this range. By default, newly created fleets have a minimum of 0 and a maximum of 1. For your auto-scaling policy to affect capacity as intended, you must increase the maximum value.

Fleet capacity is also constrained by limits on the fleet's instance type and on your AWS account. You cannot set a minimum and maximum that is outside the service and account limits.

**Track metrics after a change in capacity**

After changing capacity in response to an auto-scaling policy, Amazon GameLift waits ten minutes before responding to triggers from the same policy. This wait allows Amazon GameLift time to add the new instances, launch the game servers, connect players, and start collecting data from the new instances. During this time, Amazon GameLift continues to evaluate the policy against the metric and track the policy's evaluation period, which restarts once a scaling event is triggered. This means that a scaling policy could trigger another scaling event immediately after the wait time is over.

There is no wait time between scaling events triggered by different auto-scaling policies.

### Using Multi-Region Queues

Use Queues to create fleet groups that span multiple regions, and allow game session placement in any fleet in the queue. Queues offer numerous advantages over using individual fleets, including improved efficiency of game session placements and the ability to use player latency when selecting a game location. Client servers service specify a queue when requesting new game session placements or matchmaking. Use the AWS Command Line Interface (CLI) or the Amazon GameLift console to create, edit, and track queue metrics.

**Topics**

- Design a Game Session Queue (p. 128)
- Create a Queue (p. 133)
- View Your Queues (p. 136)

### Design a Game Session Queue

Amazon GameLift uses queues to process requests for game session placements and find hosting resources for new game sessions. The way you design your queue determines (1) where Amazon GameLift looks for available resources, and (2) how Amazon GameLift evaluates available resources to find the optimal choice for each new game session. Use queues to deliver the best possible experience to your players and to ensure that the hosting resources you're paying for are utilized efficiently.

Queues are required with these Amazon GameLift features:

- Matchmaking with FlexMatch (see Design a FlexMatch Matchmaker (p. 70))
• Spot fleets (see Spot Fleet Integration Guide (p. 103))

Learn more about queues in the section called “Running Game Sessions” (p. 5). For information on creating a queue, see Create a Queue (p. 133). For information on creating new game sessions with queues, see Create Game Sessions (p. 49).

Why Use Queues?

We highly recommend using queues for game session placement even if you're not using an Amazon GameLift feature that requires them. While you always have the option to manually create a game session on a specific fleet (CreateGameSession), queues—particularly those that span multiple regions—can offer critical benefits for you and your players.

• Minimize latency for a better player experience. When a game session placement request includes player latency data, FleetIQ ensures that players get the lowest possible latency to the game server. Set additional rules to minimize latency differences between players and to strike a balance between getting players into games fast and getting players into the best possible gaming experience.

• Take advantage of lower-priced spot fleets. You can add spot fleets, which offer significantly lower hosting costs, to a queue at any time. When spot fleets are available, FleetIQ places new game sessions in fleets with the lowest possible latency and the lowest spot price.

• Place new games faster at high capacity. A single fleet has limited capacity, and once that limit is reached, players must wait until additional instances are scaled up before new game sessions can start. A queue, however, can fall back to immediately make placements on other fleets if the preferred fleet is full. In addition, with auto-scaling, each fleet in the queue scales up as it nears full capacity, so it's unlikely that all fleets in a queue will be full at the same time. As a result, wait time for players is impacted less during surges in player demand.

• Make game availability more resilient. Fleet and region-level outages can happen. When using a multi-region queue, a slowdown or outage doesn't have to affect player access to your game. Instead, if one or more preferred fleets are unavailable, Amazon GameLift can place new game sessions with the next best fleet. Auto-scaling then adjusts for this temporary shift in fleet activity until the preferred fleet(s) are available again.

• Use extra fleet capacity more efficiently. To handle unexpected surges in player demand, it makes sense to have quick access to extra hosting capacity. When relying on a single fleet to support player access to your game, you need to maintain unused capacity just in case. In contrast, the fleets in a queue can act as fall backs, with increased player demand shifting to lesser used fleets in other regions as needed. For example, when demand is high in Asia, demand is typically low in Europe; your European fleets can provide additional capacity to support surges in Asia, even when they're scaled down during low demand.

• Get metrics on game session placements and queue performance. Amazon GameLift emits queue-specific metrics, including statistics on placement successes and failures, the number of requests in the queue, and average time that requests spend in the queue. You can view these metrics in the Amazon GameLift console or in CloudWatch.

Tips on Setting Queue Destinations

A queue contains a list of destinations where game session placement requests can be fulfilled. In most cases a destination is a fleet, which you can specify by either a fleet ID or alias ID. When choosing destinations for the queue, consider the following guidelines and best practices:

• You can add any existing fleet or alias from any region. Add a fleet in each region where you want to support players, especially if you're using player latency for placement.

• If you assign aliases to your fleets (as recommended), it is a good idea to use the alias names when setting destinations in your queue.
• All destinations in a queue must be running game builds that are compatible with the game clients that use the queue. Keep in mind that new game session requests processed by the queue might be placed on any destination in the queue.

• A queue should have at least two fleets and span at least two regions. This design improves hosting resiliency by decreasing the impact of fleet or regional slowdowns and more efficiently managing unexpected changes in player demand.

• The list order of destinations in a queue matters. If you include latency data in a game session placement request, Amazon GameLift re-prioritizes the destinations to find available resources with (1) minimal player latency and (2) lowest spot price (if relevant). If you don't provide latency data, Amazon GameLift follows the destination list order. In this situation, game sessions will usually be hosted on the first fleet listed and will only be placed on back-up fleets when needed.

• You need to decide where to create your queue (in which region). Ideally you're making game session placement requests through a game client service, such as a session directory service. If so, we recommend that you create your queue in a region that is geographically near where the client service is deployed. This arrangement minimizes latency when submitting game session placement requests.

• A queue must not have fleets with different certificate configurations. All fleets in the queue must have TLS certificate generation either enabled or disabled.

**Design a Multi-Region Queue**

A multi-region design is recommended for all queues, whether or not you're using special features such as FlexMatch matchmaking or spot fleets. This design can improve placement speed and hosting resiliency, and is critical when working with player latency data.

This topic steps through the process of designing a basic multi-region queue. For this example, we'll use a game server build that we want to distribute to players on the east coast of North America. We've chosen to host the our game in the following regions: us-east-1 and ca-central-1. We've uploaded the game build and created a fleet in each region.

1. Pick a region to create the queue in. Our queue's location may not be significant, but if we're making placement requests through a client service, we can minimize request latency by placing the queue in a region near where the client service is deployed.

2. Create a new queue and add our fleets as queue destinations.

3. Decide on a default list order for the destinations. GameLift may evaluate fleets in this order when searching for available hosting resources. If a placement request contains player latency data, Amazon GameLift reorders our destinations to prioritize the lowest latency rates. If no latency data is provided, Amazon GameLift uses the default order.

   ▪ If we expect a lot of requests without latency data, we'll want to pay more attention to our default list order and the fleets we list first. Without latency data, Amazon GameLift will always place new game sessions on the first fleet listed when available, only using the remaining fleets as backup. As a result, we'll want a fleet in the top spot that (1) is high-powered enough to handle most of our player base, and (2) is likely to deliver the highest availability and lowest latency to the majority of our players.

   In this scenario, we expect that the us-east-1 fleet will best serve most of our players, so we choose a high-performance instance type, configure it for multiple concurrent game sessions, scale for high player volume, and list it first in our queue. Our backup ca-central-1 fleet uses a smaller, less expensive instance type and maintains minimal capacity. Additional backup fleets (if added) do not need to be ordered, but we might de-prioritize regions that are likely to have high latency rates.

   ▪ If we expect most requests to have latency data, the list order is less important because FleetIQ will re-prioritize it. We'll want to have fleets in all regions that are close to our target players to minimize latency rate. Generally, we'll want fleets that are configured to handle the expected amount of player demand. We'll want to make sure that the fleet in the top spot is a good choice to handle the requests that don't have latency data.
In this scenario, we expect that both the \textit{us-east-1} and \textit{ca-central-1} fleets will provide low latency rates and be heavily used. Both fleets use high-performance instance types and are configured for multiple concurrent game sessions. Demand for the \textit{ca-central-1} fleet will be somewhat lower, so it will maintain a lower capacity.

As our queue is put into action, we can use metric data to determine how well the design is holding up. With queues, we make fleet changes as needed, either by reconfiguring existing fleets or by removing and adding new fleets that better fit our hosting needs.

**Design Player Latency Policies**

If your game session placement requests include player latency data, Amazon GameLift uses FleetIQ to ensure that players are put into game sessions with the lowest possible latency. FleetIQ prioritizes a queue's destinations based on the average regional latency for all players in the request.

You can set up player latency policies to affect how Amazon GameLift uses player latency data. You can set player latency policies to:

- Set maximum latency for individual players. Amazon GameLift by default places game sessions based on average latency across all players in the request. This policy ensures that a game session is not placed where any one player will experience latency over the maximum.
- Use multiple policies to relax maximum latency over time. Setting a maximum latency protects players from high-latency games, but they also increase the risk that a game session request with high latency values will never be fulfilled. You can use a series of policies to gradually raise the latency maximum over time. Use this approach to balance how you provide great game experiences against getting players into games with a minimum of wait time.

For example, you might define the following policies for a queue with a 5-minute timeout. Multiple policies are enforced consecutively, starting with the policy that has the lowest maximum latency value. This set of policies starts with a maximum latency of 50 milliseconds and increases it over time to 200 milliseconds. Your last policy sets the absolute maximum latency allowed for any player. If you want to ensure that all game sessions are placed regardless of latency, you can set last policy's maximum to a very high value.

1. Spend 120 seconds searching for a destination where all player latencies are less than 50 milliseconds, then...
2. Spend 120 seconds searching for a destination where all player latencies are less than 100 milliseconds, then...
3. Spend the remaining queue timeout searching for a destination where all player latencies are less than 200 milliseconds.

In this example, the first policy is in force for the first two minutes, the second policy is in force for the third and fourth minute, and the third policy is in force for the fifth minute until the placement request times out.

**Design a Queue for Spot Instances**

If you plan to use spot instances for your fleets, you'll need to set up a queue. Ideally, you want to set up a resilient queue that takes advantage of cost savings with spot fleets while minimizing the chance of game session interruptions. Use the following best practices:

- Include fleets from more than one region. A multi-region queue boosts resiliency by ensuring that there are always fleets available to host new game sessions.
In each region, include at least one spot fleet and one on-demand fleet. This design ensures that game sessions can be placed in any region even if no spot fleets are currently viable in that region.

If you include multiple spot fleets in each region, use different instance types for each fleet, preferably in the same instance family (c5.large, c5.xlarge, etc.). This design lessens the likelihood that multiple spot fleets will be unavailable or interrupted at the same time. Check the historical pricing data in the Amazon GameLift console to verify that your preferred instance types typically deliver significant cost savings with spot. This data displays pricing for spot and on-demand instances and provides estimated spot saving per instance.

To optimize FleetIQ’s ability to select the lowest priced spot instance, plan to include player latency data for all regions. As described in How FleetIQ Works (p. 132), FleetIQ works more effectively when destinations are re-prioritized by region, which is done when latency data is provided.

If you’re not providing player latency data in game session placement requests, order your destinations by preference. For example, you might list destinations by region preference (spot fleet followed by on-demand fleet). Alternatively, you might list all your spot fleets first.

### How FleetIQ Works

FleetIQ relies on the following decisionmaking process when searching for the best possible placement for a new game session.

1. FleetIQ filters the queue's destinations to remove any of the following fleets:
   - If the request includes player latency data, FleetIQ evaluates the queue's player latency policies and removes all fleets in regions where any player’s latency exceeds a policy’s maximum limit.
   - FleetIQ removes any spot fleets that are not currently viable due to unacceptable interruption rates.

2. FleetIQ prioritizes the remaining queue destinations based on the following:
   - If player latency data is provided, FleetIQ re-orders the queue destinations by region, those with lowest average player latency listed first.
   - If player latency data is not provided, FleetIQ uses the original list of queue destinations.

3. FleetIQ selects a destination from the prioritized list.
   - If the destination list was prioritized by region, FleetIQ selects the fleet in the lowest-latency region with the lowest price. If there are no viable spot fleets, any fleet in that region may be selected.
   - If the destination list was not prioritized, FleetIQ selects the first viable fleet from the original list, even if there are lower priced spot fleets on the list.

4. FleetIQ evaluates whether the selected fleet has a server process available to host a new game session. It is considered an "optimal" placement when the new game session is placed in a fleet with the lowest possible latency and/or the lowest price.

5. If the selected fleet has no available resources, FleetIQ moves to the next listed destination and repeats until it finds a fleet to host the new game session.

### Evaluate Queue Metrics

Use metrics to evaluate how well your queues are performing. You can view queue-specific metrics in the Amazon GameLift console (View Queue Details (p. 136)) or in Amazon CloudWatch. Queue metrics are described in Amazon GameLift Metrics for Queues (p. 183).

Queue metrics can provide insight in three main areas:

- **Overall queue performance** – Metrics indicate how successfully a queue is responding to placement requests and help to identify when and why placements are failing. For queues with manually scaled fleets, metrics average wait times and queue depth can indicate when capacity for a queue might need to be adjusted.
• **FleetIQ performance** – For placement requests that use FleetIQ's filtering and prioritization (that is, requests with player latency data), metrics indicate how often FleetIQ is able to find optimal placement for new game sessions. Optimal placement may include finding resources with the lowest possible player latency or, when spot fleets are available, with the lowest cost. There are also error metrics that identify common reasons why optimal placement failed.

• **Region-specific placements** – For multi-region queues, metrics show successful placements by broken down by region. With queues that use FleetIQ, this data provides useful insight into where player activity is occurring.

When evaluating metrics for FleetIQ performance, consider the following tips:

- Use the “placements succeeded” metric in combination with the FleetIQ metrics for lowest latency and/or lowest price to track the queue's rate of optimal placement.
- To boost a queue's optimal placement rate, take a look at the error metrics for the following:
  - If the error metric “first choice not available” is high, this is a good indicator that capacity scaling for the queue's fleets needs to be adjusted. It may be that all fleets in the queue are under-scaled, or there may be one particular fleet or region that is an optimal fit for most placements.
  - If the error metric “first choice not viable” is high, this is an indicator to look at your spot fleets. Spot fleets are considered “not viable” when the interruption rate for a particular instance type is too high. To resolve this issue, change the queue to use spot fleets with different instance types. As noted in the best practices for queues with spot fleets, it is always a good idea to include spot fleets with different instance types in each region.

### Create a Queue

Queues are used to place new game sessions with the best available hosting resources across multiple fleets and regions. To learn more about building queues for your game, see [Design a Game Session Queue](p. 128).

In a game client, new game sessions are started with queues by using placement requests. Learn more about game session placement in [Create Game Sessions](p. 49).

To create a queue, you can use either the Amazon GameLift console or the AWS Command Line Interface (CLI). Get and install the AWS Command Line Interface tool.

#### Create a Queue (Console)

Create a queue from the AWS Management Console.

**To create a queue**

1. Open the Amazon GameLift console at [https://console.aws.amazon.com/gamelift/](https://console.aws.amazon.com/gamelift/), and choose the region you want to create your queue in.
2. From the Amazon GameLift menu, choose **Create a queue**.
3. On the **Create queue** page, complete the **Queue Details** section:
   
   - **Queue Name** – Create a meaningful queue name so you can identify it in a list and in metrics. Requests for new a game session (using [StartGameSessionPlacement](https://docs.aws.amazon.com/gamelift/latest/ug/using-content.html#start-game-session-placement)) must specify a queue by this name. Spaces and special characters are not allowed.
   
   - **Queue Timeout** – Specify how long you want Amazon GameLift to try to place a new game session before stopping. Amazon GameLift continues to search for available resources on any fleet until the request times out.
4. Under **Player latency policies**, define zero or more policies for the queue. For each placement request, Amazon GameLift automatically minimizes the average latency across all players. You
can also create a latency policies to set a maximum limit for each individual player. Player latency policies are evaluated only when player latency data is provided in the placement request. You can opt to enforce one limit throughout the placement process, or you can gradually relax the limit over time. Learn more at Design Player Latency Policies (p. 131).

a. To add a first policy, choose Add player latency policy. Enter a maximum player latency value for this policy (default is 150 milliseconds). As indicated in the policy language, this first policy will be enforced either for the entire placement process or—if you create additional policies—for any remaining time after the other policies have expired.

b. To add another player latency policy, choose Add player latency policy again. For additional policies, set the maximum player latency value and a length of time (in seconds) to enforce it. Maximum latency values for these policies must be lower than the first policy.

As you add policies, the console automatically reorders the policies based on the maximum player latency value, lowest value listed first. This is the order in which the policies are enforced during a game session placement effort.

5. Under Destinations, add one or more destinations to the queue. A queue can contain fleets from multiple regions and with both on-demand and spot fleets. All fleets in the queue must have the same certificate configuration (either GENERATED or DISABLED). All fleets should be running game builds that are compatible with the game clients that will use the queue, such as fleets in multiple regions that are running the same game build. Fleets and aliases must exist before you can add them as destinations.

   a. Choose Add destination.

   b. Use the columns to specify the region and type (fleet or alias) for your destination. From the resulting list of fleet or alias names, select the one you want to add.

   c. To save the destination, choose the green checkmark icon. You must save each destination before adding another one, changing the default order, or saving the queue.

   d. If you have multiple destinations, set the default order by using the arrow icons in the Priority (default) column. This order is used by Amazon GameLift when searching destinations for available resources to place a new game session. (The default order is overridden if a game session placement request includes player latency data.)

6. Once you've finished configuring your new queue, choose Create queue. Your new queue is saved and the Queues page shows the new queue and any other queues that exist. You can choose a queue from this page to view detailed information, including queue metrics. You can edit a queue configuration at any time.

Create a Queue (AWS CLI)

You can use the AWS Command Line Interface (AWS CLI) to create a queue. Get and install the AWS Command Line Interface tool.

To create a queue

- Open a command line window and use the create-game-session-queue command to define a new queue. For more information, see the AWS CLI Command Reference.

The following example creates a queue for placing new game sessions on any of several fleets before timing out at five minutes. Fleets are listed as destinations and identified by either a fleet ARN or alias ARN. All fleets and aliases must already exist. Amazon GameLift tries to place new game sessions on fleets in the order they are listed here unless the order is overridden by individual game session placement requests.
Note
You can get fleet and alias ARN values by calling either `describe-fleet-attributes` or `describe-alias` with the fleet or alias ID. For more information on ARN (Amazon Resource Name) formats, see ARNs and AWS Service Namespaces.

```bash
$ aws gamelift create-game-session-queue
   --name "Sample test queue"
   --timeout-in-seconds 300
   --destinations DestinationArn=arn:aws:gamelift:us-east-1::alias/alias-a1234567-b8c9-0d1e-2fa3-b45c6d7e8910
   DestinationArn=arn:aws:gamelift:us-west-2::alias/alias-b0234567-c8d9-0e1f-2ab3-c45d6e7f8901
   DestinationArn=arn:aws:gamelift:us-west-2::fleet/fleet-f1234567-b8c9-0d1e-2fa3-b45c6d7e8912
```

Copiable version:

```bash
aws gamelift create-game-session-queue --name "Sample test queue" --timeout-in-seconds 300
   --destinations DestinationArn=arn:aws:gamelift:us-east-1::alias/alias-a1234567-b8c9-0d1e-2fa3-b45c6d7e8910
   DestinationArn=arn:aws:gamelift:us-west-2::alias/alias-b0234567-c8d9-0e1f-2ab3-c45d6e7f8901
   DestinationArn=arn:aws:gamelift:us-west-2::fleet/fleet-f1234567-b8c9-0d1e-2fa3-b45c6d7e8912
```

If the `create-game-session-queue` request is successful, Amazon GameLift returns a `GameSessionQueue` object with the new queue configuration. You can now submit requests to the queue using `StartGameSessionPlacement`.

To create a queue with player latency policies

- Open a command line window and use the `create-game-session-queue` command to define a new queue. For more information, see the AWS CLI Command Reference.

The following example creates a queue with a 10-minute timeout, three destinations, and a set of player latency policies. In this example, the first player latency policy is in force for the first two minutes, the second policy is in force for the third and fourth minute, and the third policy is in force for six minutes until the placement request times out.

```bash
$ aws gamelift create-game-session-queue
   --name "matchmaker-queue"
   --timeout-in-seconds 600
   --destinations DestinationArn=arn:aws:gamelift:us-east-1::alias/alias-a1234567-b8c9-0d1e-2fa3-b45c6d7e8910
   DestinationArn=arn:aws:gamelift:us-west-2::alias/alias-b0234567-c8d9-0e1f-2ab3-c45d6e7f8901
   DestinationArn=arn:aws:gamelift:us-west-2::fleet/fleet-f1234567-b8c9-0d1e-2fa3-b45c6d7e8912
   --player-latency-policies
   "MaximumIndividualPlayerLatencyMilliseconds=50,PolicyDurationSeconds=120"
   "MaximumIndividualPlayerLatencyMilliseconds=100,PolicyDurationSeconds=120"
   "MaximumIndividualPlayerLatencyMilliseconds=150"
```

Copiable version:

```bash
aws gamelift create-game-session-queue --name "matchmaker-queue" --timeout-in-seconds 600
   --destinations DestinationArn=arn:aws:gamelift:us-east-1::alias/alias-a1234567-b8c9-0d1e-2fa3-b45c6d7e8910
   DestinationArn=arn:aws:gamelift:us-west-2::alias/alias-b0234567-c8d9-0e1f-2ab3-c45d6e7f8901
   DestinationArn=arn:aws:gamelift:us-west-2::fleet/fleet-f1234567-b8c9-0d1e-2fa3-b45c6d7e8912
   --player-latency-policies
   "MaximumIndividualPlayerLatencyMilliseconds=50,PolicyDurationSeconds=120"
   "MaximumIndividualPlayerLatencyMilliseconds=100,PolicyDurationSeconds=120"
   "MaximumIndividualPlayerLatencyMilliseconds=150"
```
If the `create-game-session-queue` request is successful, Amazon GameLift returns a `GameSessionQueue` object with the new queue configuration.

**View Your Queues**

You can view information on all existing game session placement queues (p. 128). Queues shown include only those created in the selected region. From the `Queues` page, you can create a new queue, delete existing queues, or open a details page for a selected queue. A `Queue` detail page contains the queue's configuration and metrics data. You can also edit or delete the queue.

**To view the Queues page**

1. Choose `Queues` from the Amazon GameLift console's menu.

   The `Queues` page displays the following summary information for each queue:
   - **Queue name** – The name assigned to the queue. Requests for new game sessions specify a queue by this name.
   - **Queue timeout** – Maximum length of time, in seconds, that a game session placement request remains in the queue before timing out.
   - **Destinations in queue** – Number of fleets listed in the queue configuration. New game sessions can be placed on any fleet in the queue.

2. To view details for a queue, including metrics, choose the queue name. For more information on the queue details page, see `View Queue Details` (p. 136).

**View Queue Details**

You can access detailed information on any queue, including the queue configuration and metrics. To open a `Queue` detail page, go to the main `Queues` page and choose a queue name.

The queue detail page displays a summary table and tabs containing additional information. On this page you can do the following:

- Update the queue's configuration, list of destinations and player latency policies. Choose **Actions, Edit queue**.
- Remove a queue. After a queue is removed, all requests for new game sessions that reference that queue name will fail. (Note that deleted queues can be restored by simply creating a queue with the deleted queue's name.) Choose **Actions, Delete queue**.

**Summary**

The summary table includes the following information:

- **Queue name** – The name assigned to the queue. Requests for new game sessions specify a queue by this name.
- **Queue timeout** – Maximum length of time, in seconds, that a game session placement request remains in the queue before timing out.
- **Destinations in queue** – Number of fleets listed in the queue configuration. New game sessions can be placed on any fleet in the queue.
Destinations

The **Destinations** tab shows all fleets or aliases listed for the queue. Fleets are identified by either a fleet ARN or an alias ARN, which specifies the fleet or alias ID and the region.

When Amazon GameLift searches the destinations for available resources to host a new game session, it searches in the order listed here. As long as there is capacity on the first destination listed, new game sessions are placed there. This is the default order for destinations. You can have individual game session placement requests override the default order by providing player latency data. This data tells Amazon GameLift to search for an available destination with the lowest average player latency.

To add, edit, or delete destinations, choose **Actions, Edit queue**.

Player Latency Policies

The **Player latency policies** tab shows all policies that have been defined for the queue. Policies are listed in the order they are enforced during a game session placement effort.

To add, edit, or delete player latency policies, choose **Actions, Edit queue**.

Queue Metrics

The **Metrics** tab shows a graphical representation of queue metrics over time.

Queue metrics include a range of information describing placement activity across the entire queue, as well as successful placements broken down by region. The region-specific data is useful for tracking where your games are being hosted. With queues that use FleetIQ and prioritize placements to minimize player latency and hosting costs, regional placement metrics can help to detect issues with the overall queue design.

Queue metrics are also available in Amazon CloudWatch. You can view the descriptions of all metrics at [Amazon GameLift Metrics for Queues (p. 183)](https://docs.aws.amazon.com/gamelift/latest/developerguide/gameli.htm).

**To display metrics information in the graph**

1. Click one or more of the metric names that are listed to the left of the graph area. Metric names shown in black are displayed in the graph, while those shown in gray are turned off. Use the color key to identify which graphed line matches a selected metric.
2. Use the following filters, shown above the graph area, to change how metric data is displayed:
   - **Data and Period** – Offers two options for selecting a date range:
     - Use **Relative** to select a period of time relative to the current time, such as Last hour, Last day, Last week.
     - Use **Absolute** to specify a period with an arbitrary start and end date/time.
   - **Granularity** – Select a length of time to aggregate data points.
   - **Refresh rate** – Select how often you want the graph display to be updated. You can refresh the graph any time by clicking the refresh button in the graph's upper right corner.
   - **Time zone** – Select which time format to use in the graph display: UTC (universal coordinated time) or Browser time (local time).
   - **Show points** – Toggle to display discrete data points as circles or display lines only.
Setting Up Amazon GameLift FlexMatch Matchmakers

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 Matchmaking Process (p. 14).

Topics

- Design a FlexMatch Matchmaker (p. 138)
- Create a Matchmaking Configuration (p. 139)
- Build a FlexMatch Rule Set (p. 141)
- Set up FlexMatch Event Notification (p. 163)

Design a FlexMatch Matchmaker

Matchmaker Configurations

At a minimum, a matchmaker needs three 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. For more information on creating rule sets, see Build a FlexMatch Rule Set (p. 141).

- The game session queue determines where matches created by this matchmaker will be hosted. The matchmaker uses the queue to find available resources and to place a game session for the match. A queue specifies which regions to use when placing a game session. For more information on creating queues, see Create a Queue (p. 133).

- 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.

In addition to these minimum requirements, you can configure your matchmaker with the following additional options.

Player Acceptance

You can configure a matchmaker to require that all players selected for a match must accept participation. If acceptance is required, 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.

The matchmaking requests are handled as follows: requests where all players accepted the match are returned to the matchmaking pool for continued processing; requests where at least one player rejected the match or failed to respond fail and are no longer processed. Player acceptance requires a time limit, which you can define; all players must accept a proposed match within the time limit for the match to continue.

Matchmaking Notification
This feature emits all matchmaking-related events to a designated Amazon Simple Notification Service (SNS) topic. Since matchmaking requests are asynchronous, all games must have a way to track the status of matchmaking requests, and setting up notifications is simple and efficient. See more on options for tracking requests in Add FlexMatch to a Game Client (p. 70). To use notifications, you first need to set up an SNS topic, and then provide the topic ARN as the notification target in the matchmaking configuration. See more information on setting up notifications at Set up FlexMatch Event Notification (p. 163).

Backfill Mode

FlexMatch backfill is useful 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 and the same process to find new players as was used to match the original players. By setting the backfill mode in the matchmaking configuration, you can opt to use automatic backfill or manage backfill requests manually. Learn more about automatic backfill in Backfill Existing Games with FlexMatch (p. 76).

With backfill mode set to AUTOMATIC, GameLift triggers a new backfill request whenever a game session has an open player slot. GameLift begins generating backfill requests for any game session that has open slots as soon as the game session starts up on a game server. Automatic backfill continues unless it is explicitly turned off for a game session. If your matchmaker uses a rule set that defines large matches (greater than 40 players), backfill requests that are generated automatically are given a higher priority. This means that, as new players arrive and request a game slot, they are more likely to be placed in an existing game than in a brand new game.

Set backfill mode to MANUAL if you do not want to backfill your games, or if you want to manually trigger backfill requests for your games. Manual backfill gives you the flexibility to decide when to trigger a backfill request. For example, you may not want to add new players during certain phases of your game or only when certain conditions exist. You can manually trigger backfill requests from either your game server (recommended because the game server has the most up-to-date player counts) or from your game client.

Game Properties

When requesting a new game session, you can pass additional information to your game server about the type of game session to create. Game properties can be delivered with any type of game session request--with a placement request or with a matchmaking request. With matchmaking, game properties are included in the matchmaker configuration. All game sessions created using a matchmaker use the same game properties. If you need to vary game properties, create separate matchmakers and send matchmaking requests to the matchmaker with the appropriate game properties.

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 FlexMatch matchmaker, you create a new matchmaking configuration. Use either the Amazon GameLift console or the AWS Command Line Interface (AWS CLI). For more detailed information on configuring a matchmaker, see the section called “Design a Matchmaker” (p. 138).
Choose a Region for the Matchmaker

A matchmaker is hosted in the region where its configuration is created. It's worth considering how the location of a matchmaker might affect its performance and how to refine the match experience for its intended players. We recommend that you place a matchmaker in a region close to the clients or client service that will be requesting matchmaking. As a best practice, we also recommend putting the matchmaker and the queue it uses in the same region. This helps to minimize communication latency between the matchmaker and queue.

Create a Matchmaker

Before you can create a matchmaking configuration, you must create the rule set and queue that you want to use with the matchmaker.

Console

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/home.
2. Switch to the region where you want to place your matchmaker.
3. From the main menu, choose Create matchmaking configuration. Fill in the matchmaking configuration details.
   - **Name** – Create a meaningful matchmaker name so you can easily identify it in a list and in metrics. The matchmaker name must be unique within a region. Matchmaking requests identify which matchmaker to use by its name and region.
   - **Description** – (Optional) Add a description of the matchmaker. The description is used for identification only; it is not used in the matchmaking process.
   - **Queue** – Choose the game session queue to use with this matchmaker. To find the queue, first choose the region where the queue is configured. Then choose the queue you want from the list of available queues in that region.
   - **Request timeout** – Type the maximum amount of time, in seconds, for the matchmaker to complete a match for each request. Matchmaking requests that exceed this time are terminated.
   - **Acceptance required** – (Optional) Indicate whether to require each player in a proposed match to actively accept participation in the match. If you chose yes, indicate how long you want the matchmaker to wait for player acceptances before canceling the match.
   - **Rule set name** – Choose the rule set to use with this matchmaker. The list contains all rule sets that have been created in the current region.
   - **Backfill mode** – Specify a method for handling match backfills. Select "Automatic" to turn on the automatic backfill feature. Or select "Manual" if you are managing backfill requests in your game server or game client, or if you opt to not backfill your games.
   - **Notification target** – (Optional) Type the ARN of an SNS topic for receiving matchmaking event notifications. If you haven't set one up yet, you can add this information later by editing the matchmaking configuration. See Set up FlexMatch Event Notification (p. 163)
   - **Additional players** – (Optional) Specify the number of player slots to remain unfilled in each new match. These slots can be filled with players in the future.
   - **Custom event data** – (Optional) Specify any data you want to associate with this matchmaker in event messaging. This data is included in every event that is associated with the matchmaker.
4. Once you've finished configuring a matchmaker, click Create. If the creation is successful, the matchmaker is immediately 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. See complete
documentation on this command in the AWS CLI Command Reference. Get and install the AWS Command Line Interface tool.

This example creates a new matchmaking configuration that requires player acceptance and uses notifications to track the status of matchmaking requests. It also reserves two player slots for additional players to be added later.

```
$ aws gamelift create-matchmaking-configuration
--name "SampleMatchmaker123"
--description "The sample test matchmaker with acceptance"
--rule-set-name "My_Rule_Set_One"
--request-timeout-seconds "120"
--acceptance-required "true"
--acceptance-timeout-seconds "30"
--backfill-mode "AUTOMATIC"
--additional-player-count "2"
--game-session-data "key=map,value=winter444"
```

Copiable version:

```
aws gamelift create-matchmaking-configuration
```

If the matchmaking configuration creation request is successful, Amazon GameLift returns a MatchmakingConfiguration object with the settings that you requested for the matchmaker. The new matchmaker is immediately 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 within 10 points. 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 (p. 142)
- Create Matchmaking Rule Sets (p. 147)
- FlexMatch Rule Set Examples (p. 148)
- FlexMatch Rules Language (p. 276)
Design a FlexMatch Rule Set

At its most basic, a matchmaking rule set does two things: it lays out a match's team structure and size, and it tells the matchmaker how to evaluate players to find the best possible match. But it can do quite a bit more than that. For example, rule set is also where you address matchmaking issues such as these:

- Trigger special match processing for large matches (>40 players).
- Enforce minimum player latency requirements to protect players' gameplay.
- Relax team requirements or match rules gradually if no match can be made.
- Define special handling for match requests that contain multiple players (party aggregation).

This topic covers the basic structure of a rule set and how to use them for matches with up to 40 players. Matches for greater than 40 players require a different rule set structure, because FlexMatch uses a streamlined algorithm to quickly match large groups of players. Learn more about building large matches in Design a FlexMatch Large-Match Rule Set (p. 144).

Related Topics

- Create Matchmaking Rule Sets (p. 147)
- FlexMatch Rule Set Examples (p. 148)
- FlexMatch Rules Language (p. 276)

Define Rule Set Components

All rule sets specify some or all of the following components. At a minimum, a rule set must specify the rule language version, define a team, and set one rule. In most cases, you also want to declare a player attribute and use it in a custom rule.

Review the general rule set schema in Rule Set Schema for Large Matches (p. 275).

There are additional requirements for a rule set that creates large matches (greater than 40 players). Learn more about large matches in Design a FlexMatch Large-Match Rule Set (p. 144).

Describe Rule Set

Provide details for the rule set.

- name (optional) – This is a descriptive label within the rule set syntax and is not used by Amazon GameLift in any meaningful way. Do not confuse this value with the rule set name, which is set, along with the rule set syntax, when you create a rule set.

- ruleLanguageVersion (required) – This is the version of the property expression language used to create FlexMatch rules. The value must be equal to “1.0”.

Declare Player Attributes

Rules may choose players for matches based on individual player characteristics. If you create rules that rely on player attributes, they must be declared in this section. Values for declared player attributes should be included in every matchmaking request that is sent a matchmaker using this rule set.

You may also want to pass certain player attributes to the game session even if the rule set doesn't use them during player evaluation. For example, you might pass a player's character choice. To do this, declare your player attributes here, and include the attribute values for each player in your matchmaking requests.
When declaring a player attribute, include the following information:

- **name** (required) – This value must be unique to the rule set.
- **type** (required) – This is the data type of the attribute value. Valid data types are number, string, or string map.
- **default** (optional) – Enter a default value to use when no value is provided for a player. If no default is declared and a player does not provide a value, the player cannot be matched.

**Define 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, while 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. This name is used in rules and expansions, and it is referenced in the matchmaking data that is used in the game session.
- **maxPlayers** (required) – Specify the maximum number of players that can be assigned to the team.
- **minPlayers** (required) – Specify the minimum number of players that must be assigned to the team before the match can succeed.
- **quantity** (optional) – If you want FlexMatch to create more than one team based on this definition, specify how many. When FlexMatch creates a match, these teams are given the designated name with an appended number. For example “Red-Team_1”, “Red-Team_2”, “Red-Team_3”, etc.

FlexMatch always tries to fill teams to the maximum player size but does create teams with fewer players when the minimum player size allows it. 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. 148) topic for an example of an "EqualTeamSizes" rule.

**Set Rules for Player Matching**

Create a set of rule statements that define how to 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.

- **name** (required) – This is 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. This information is not used by the matchmaker.
- **type** (required) – The type element identifies the operation to use when processing the rule. Each rule type requires a set of additional properties. For example, several rule types need a reference value to measure a player's attributes against. See a list of valid rule types and properties in FlexMatch Rules Language (p. 276).
- **Rule type property (may be required)** – Depending on the type of rule being defined, you may need to set certain rule properties. For example, with distance and comparison rules, you must specify which player attribute to measure. Learn more about properties and how to use the FlexMatch property expression language in FlexMatch Rules Language (p. 276).
Relax Match Requirements Over Time

Expansions allow you to relax match criteria over time when no valid matches are possible. This feature ensures that a “best available” match can be made when a perfect match is not possible. You might use an expansion to relax a player skill requirement, increase acceptable player latency levels, or decrease the minimum number of players required. By relaxing your rules with an expansion, you are gradually expanding the pool of players that can be matched.

- **target** (required) – Identify the rule set element to be relaxed. You can relax a rule or a team property.
- **steps** (required) – You can relax a rule in multiple stages. For each step, specify the wait time and the new value to apply. For rule set expansions, wait times are absolute, representing a number of seconds after match creations starts. For expansions with multiple steps, each step’s wait time must be larger than the previous step. The last step’s wait time cannot be longer than the time allowed for a match request, which is set in the matchmaking configuration.
  - **waitTimeSeconds**
  - **value**

If this rule set is used by a matchmaker that has 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. If you have very short wait times in your expansion steps, FlexMatch is likely to create a lot of partially-filled matches before new games start backfilling. A better approach is to set your expansion wait times only after automatic backfill tends to kicks in for your games. This helps FlexMatch get players into games (new or existing) faster and more efficiently. At the same time, you want FlexMatch to backfill your existing games (using this same rule set) as quickly as possible. Expansion timing varies depending on your team composition. Expect to do some testing to find the best expansion strategy for your game.

Design a FlexMatch Large-Match Rule Set

If your rule set can create matches with more than 40 players, FlexMatch processes match requests that use that rule set as large matches. Large matches are processed using a different algorithm that significantly reduces the amount of time required to match high numbers of players.

To determine whether your rule set creates large matches, look at the `maxPlayer` setting for all teams in your rule set. If the total of these settings exceeds 40, you’ve got a large match rule set. Large match rule sets can create matches up to 200 players.

A large match rule set uses the same components as other rule sets, with a few adjustments. In addition, the rule set must include an algorithm component. Review the schema for a large match rule set in Rule Set Schema for Large Matches (p. 275).

Define Large Match Algorithm

Add an algorithm component to the rule set. This component configures the large-match algorithm for your preferences.

- **batchingPreference** (required) – This property indicates the way that players are sorted during match creation. The value must be equal to “balanced”.

  **balancedAttribute** (required) – Identify a single player attribute to use when choosing players for a match. Before evaluating individual players, FlexMatch sorts the available player pool according to this attribute. It starts by evaluating players that have similar attribute values, and gradually moves on to players that are less similar until the match is filled. This attribute is used to achieve a player balance in the match, grouping players that tend to have similar values for this attribute. For example, if you choose a skill attribute, players with similar skill levels are grouped together in a match. This mechanism is most effective for large matches when you have large pools of available players.
Be sure to declare the balancing attribute in the rule set's player attributes. Only attributes with data type "number" can be used as a balancing attribute.

- **strategy** (required) – Choose a matching strategy to use during match creation. Options are "largestPopulation" (the default) and "fastestRegion".

### Largest population

With this strategy, FlexMatch maintains the largest possible player pool by including all players who have acceptable latency values in at least one region. With a large player pool, matches tend to fill more quickly, and matched players are more similar with regard to the balancing attribute. Players may be placed in games where their latency is less than ideal, although still within acceptable limits.

### Fastest region

This strategy places a priority on getting players into matches that deliver the best possible latency for them. FlexMatch groups available players based on the regions where they report lowest latency values, and then tries to fill matches from these groups. FlexMatch favors placing players in the fastest possible regions for them. However, it may group players based on their second- or third-fastest regions (or slower) in order to create groups large enough to fill a match. As a result, matches may take longer to fill. In addition, players in a match that is created with this strategy may vary more widely with regard to the balancing attribute.

Here's an example:

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

### Declare Player Attributes

At a minimum, you must declare the player attribute that is used as a balancing attribute in the rule set algorithm. Only attributes with data type "number" can be used as the balancing attribute.

In addition, you may want to pass certain player attributes to the game server to use when setting up the game session. For example, you could pass a player's character choice, map preference, etc. To pass on player attributes, declare them in the rule set and then include attribute values for each player in your matchmaking requests. When GameLift passes the game session request to the game server, it includes the matchmaker data, which contains attribute values for all matched players.

### 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 what matches are likely to look like when only partially filled. You may want to alter 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, matches have similar numbers of players assigned to each team, even if 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 initially get assigned to the largest teams with the most open slots. Then, as the number of open slots become more evenly distributed across all teams, players start getting added to the smaller teams.
Build a Rule Set

Let's walk through an example. 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 sequence for filling this match:

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.
3. All teams have now reached minPlayers. Red and Blue teams have the most open slots, so the next 8 players are assigned (4 each) to the Red and Blue teams.
4. Once all three teams have 1 open slot available, the remaining 3 player slots are assigned in no particular order.

Set Latency Rule for Large Matches

Since most of the work of creating large matches is done using the balancing player attribute and prioritization strategy. Most custom rules are not available. However, you can create a rule that sets a hard limit on player latency.

To create this rule, use the `latency` rule type with the property `maxLatency`. Here's an example that sets maximum player latency to 200 milliseconds:

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

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 latency rules or 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.

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

Manage matchmaking rule sets for your FlexMatch matchmakers. Use either the Amazon GameLift console or the AWS Command Line Interface (CLI). Learn more about how FlexMatch matchmaking works in the section called "How GameLift FlexMatch Works" (p. 13).

Once created, matchmaking rule sets cannot be changed, so we recommend checking the rule set syntax before creating the rule set. Both the console and the AWS CLI provide a validation option. There is a maximum limit on the number of rule sets you can have, so it's a good idea to delete unused rule sets.

Console

To create a rule set:

1. Open the Amazon GameLift console at https://console.aws.amazon.com/gamelift/.
2. Switch to the region where you want to place your rule set. Rule sets must be defined in the same region as the matchmaking configuration they will be used with.
3. From the Amazon GameLift main menu, choose Create matchmaking rule set and fill in the rule set details.
   - **Rule set name** – Create a meaningful name so you can easily identify it in a list and in events and metrics. The rule set name must be unique within a region. Matchmaking configurations identify which rule set to use by its name. Note: This is not the same as the "name" field in the rule set body, which is not currently used.
   - **Rule set** – Enter the JSON text of a rule set body. Learn more about designing a rule set in the section called “Design a Rule Set” (p. 142), or use one of the example rule sets from FlexMatch Rule Set Examples (p. 148).
4. Since rule sets can't be edited once they're created, it's a good idea to validate your rule set first. Click Validate rule set to verify that the syntax of your rule set body is correct.
5. Once you've finished configuring a matchmaker, click Create rule set. If creation is successful, the rule set can be used by a matchmaker.

To delete a rule set:

1. On the Matchmaking rule sets console page, select a rule set and Click Delete rule set.
2. If the rule set being deleted is currently being used by a matchmaking configuration, an error message is displayed. In this case, you must change the matchmaking configuration to use a different rule set before you can delete the rule set. To find out which matchmaking configurations are currently using a rule set, click on the rule set name to view the rule set's detail page.

AWS CLI

To create a rule set:

To create a matchmaking rule set with the AWS CLI, open a command line window and use the command create-matching-rule-set (AWS CLI Command Reference. Get and install the AWS Command Line Interface tool).

This example creates a simple matchmaking rule set that sets up a single team. Be sure to create the rule set in the same region as the matchmaking configurations that will reference it.

```bash
$ aws gamelift create-matching-rule-set
```
Build a Rule Set

--name "SampleRuleSet123"
--rule-set-body '{"name": "aliens_vs_cowboys",
"ruleLanguageVersion": "1.0",
"teams": [{
"name": "cowboys",
"maxPlayers": 8,
"minPlayers": 4}]}'

Copiable version:

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, Amazon GameLift returns a MatchmakingRuleSet object that includes the settings you specified. The new rule set can now be used by a matchmaker.

To delete a rule set:

- To delete a matchmaking rule set with the AWS CLI, open a command line window and use the command delete-matchmaking-rule-set (AWS CLI Command Reference).

If the rule set being deleted is currently being used by a matchmaking configuration, an error message is displayed. In this case, you must change the matchmaking configuration to use a different rule set before you can delete the rule set. To get a list of which matchmaking configurations are currently using a rule set, use the command describe-matchmaking-configurations (AWS CLI Command Reference) and specify the rule set name.

This example first 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"

Copiable versions:

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

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

If the delete request is successful, Amazon GameLift returns success.

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:
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
calculated 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".

```json
{
  "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",
```
Build a Rule Set

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.

```json

{  
  "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  
    }]  
  }],  
}

Version 151
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.

```json
{
  "name": "three_team_game",
  "ruleLanguageVersion": "1.0",
  "playerAttributes": [
    {
      "name": "skill",
      "type": "number",
      "default": 10
    },
    {
      "name": "character",
      "type": "string_list",
      "default": [ "peasant" ]
    }
  ]
}
```
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 presort 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": [ {
        "name": "MapPreference",
        "description": "Favor grouping players that have the highest map preference aligned with the anchor's favorite",
```

Version
154
// 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])"]
},
"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.

```json
{
  "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": [
```
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 must 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".

```json
{
    "waitTimeSeconds": 15,
    "value": 0
}
}
```
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 If your rule set can create matches with more than 40 players, FlexMatch processes match requests that use that rule set as large matches. Large matches are processed using a different algorithm that significantly reduces the amount of time required to match high numbers of players.

To determine whether your rule set creates large matches, look at the maxPlayer setting for all teams in your rule set. If the total of these settings exceeds 40, you've got a large match rule set. Large match rule sets can create matches up to 200 players.

A large match rule set uses the same components as other rule sets, with a few adjustments. In addition, the rule set must include an algorithm component. Review the schema for a large match rule set in Rule Set Schema for Large Matches (p. 275).

Define Large Match Algorithm

Add an algorithm component to the rule set. This component configures the large-match algorithm for your preferences.

- **batchingPreference** (required) – This property indicates the way that players are sorted during match creation. The value must be equal to “balanced”.
- **balancedAttribute** (required) – Identify a single player attribute to use when choosing players for a match. Before evaluating individual players, FlexMatch sorts the available player pool according to this attribute. It starts by evaluating players that have similar attribute values, and gradually moves on to players that are less similar until the match is filled. This attribute is used to achieve a player balance in the match, grouping players that tend to have similar values for this attribute. For example, if you choose a skill attribute, players with similar skill levels are grouped together in a match. This mechanism is most effective for large matches when you have large pools of available players. Be sure to declare the balancing attribute in the rule set's player attributes. Only attributes with data type "number" can be used as a balancing attribute.
- **strategy** (required) – Choose a matching strategy to use during match creation. Options are "largestPopulation" (the default) and "fastestRegion".
Largest population

With this strategy, FlexMatch maintains the largest possible player pool by including all players who have acceptable latency values in at least one region. With a large player pool, matches tend to fill more quickly, and matched players are more similar with regard to the balancing attribute. Players may be placed in games where their latency is less than ideal, although still within acceptable limits.

Fastest region

This strategy places a priority on getting players into matches that deliver the best possible latency for them. FlexMatch groups available players based on the regions where they report lowest latency values, and then tries to fill matches from these groups. FlexMatch favors placing players in the fastest possible regions for them. However, it may group players based on their second- or third-fastest regions (or slower) in order to create groups large enough to fill a match. As a result, matches may take longer to fill. In addition, players in a match that is created with this strategy may vary more widely with regard to the balancing attribute.

Here's an example:

```
"algorithm": {

  "balancedAttribute": "player_skill",

  "strategy": "balanced",

  "batchingPreference": "largestPopulation"

},
```

Declare Player Attributes

At a minimum, you must declare the player attribute that is used as a balancing attribute in the rule set algorithm. Only attributes with data type "number" can be used as the balancing attribute.

In addition, you may want to pass certain player attributes to the game server to use when setting up the game session. For example, you could pass a player's character choice, map preference, etc. To pass on player attributes, declare them in the rule set and then include attribute values for each player in your matchmaking requests. When GameLift passes the game session request to the game server, it includes the matchmaker data, which contains attribute values for all matched players.

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 what matches are likely to look like when only partially filled. You may want to alter 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, matches have similar numbers of players assigned to each team, even if 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 initially get assigned to the largest teams with the most open slots. Then, as the number of open slots become more evenly distributed across all teams, players start getting added to the smaller teams.
Let's walk through an example. 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 sequence for filling this match:

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.

3. All teams have now reached minPlayers. Red and Blue teams have the most open slots, so the next 8 players are assigned (4 each) to the Red and Blue teams.

4. Once all three teams have 1 open slot available, the remaining 3 player slots are assigned in no particular order.

Set Latency Rule for Large Matches

Since most of the work of creating large matches is done using the balancing player attribute and prioritization strategy. Most custom rules are not available. However, you can create a rule that sets a hard limit on player latency.

To create this rule, use the latency rule type with the property maxLatency. Here's an example that sets maximum player latency to 200 milliseconds:

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

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 latency rules or 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": 6
    }]
},
```
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. 146).
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.
  \(10 \times 15 \text{ hunters} + 5 \text{ monsters} = 155\)
- 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.
  - After 15 seconds, 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. 146).

```json
{
    "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
        }]
    }]
}
```

**Set up FlexMatch Event Notification**

If you're using FlexMatch matchmaking in your game, you need a way to track the status of individual matchmaking requests and take action as appropriate. Sometimes, such as when players are required to accept a proposed match, these actions are time sensitive. The simplest method for tracking each request's status is to continuously poll using DescribeMatchmaking. However, for a faster solution, we recommend using notifications to track matchmaking events. Event notifications are simple to set up and are significantly faster and more efficient.

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. Alternatively, you can set up your own SNS topic(s) and attach them to your matchmaker to receive matchmaking event notifications directly.
Set up Event Notification

See the list of FlexMatch events emitted by Amazon GameLift in FlexMatch Matchmaking Events (p. 280).

Set up CloudWatch Events

Amazon 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. Learn more about how to use CloudWatch Events in the Getting Started guide, which includes a collection of tutorials.

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

To access Amazon GameLift matchmaking events in CloudWatch Events, go to the Amazon CloudWatch console and open Events. Be sure that you're in the region where you've set up your matchmaking configuration. For more information about getting account credentials to access CloudWatch Events, see Sign in to the Amazon CloudWatch Console. Each matchmaking event is identified by the service (GameLift), the matchmaking name, and the matchmaking ticket.

Set up an SNS Topic

You can ask Amazon GameLift to publish all events generated by a FlexMatch matchmaker to an Amazon Simple Notification Service (SNS) topic. When configuring the matchmaker, set the notification target field to an SNS topic ARN.

To set up an SNS topic for Amazon GameLift event notification

1. Go to the Amazon Simple Notification Service console.
2. Create a topic. From the SNS dashboard, choose Create topic and follow the instructions to create your topic. When the topic is created, the console automatically opens the Topic details page for the new topic.
3. Allow Amazon GameLift to publish to the topic. If you're not already in the Topic details page for your topic, choose Topics from the navigation bar and click the topics ARN to open it. Choose the topic action Edit topic policy, and go to the Advanced view tab.

Add the bolded syntax below to the end of your existing policy. (The entire policy is shown for clarity.)

```json
{
  "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",
        "SNS:Receive"
      ]
    }
  ]
}
```
VPC Peering for Amazon GameLift

This topic provides guidance on how to set up a VPC peering connection between your GameLift-hosted game servers and your other non-GameLift resources. Use Amazon Virtual Private Cloud (VPC) peering connections to enable your game servers to communicate directly and privately with your other AWS resources, such as a web service or a repository.

**Note**

VPC peering is an advanced feature. Learn about alternative options for enabling your game servers to communicate directly and privately with your other AWS resources at Access AWS Resources From Your Fleets (p. 45).

If you’re already familiar with Amazon VPCs and VPC peering, please note that setting up peering with GameLift game servers is somewhat different. You don’t have access to the VPC for your game servers (it is controlled by the GameLift service), so you can’t directly request VPC peering for it. Instead, you first pre-authorize the VPC with your non-GameLift resources to accept a peering request from the GameLift service. You then trigger GameLift to request the VPC peering that you just authorized. GameLift creates the peering connection, sets up the route tables, and configures the connection.

**Set up VPC Peering for an Existing Fleet**

**To set up a VPC peering connection:**

1. **Get AWS Account ID(s) and credentials.**

   You need an ID and sign-in credentials for the following AWS accounts. You can find AWS account IDs by signing into the AWS Management Console and viewing your account settings. To get credentials, go to the IAM console.

   - AWS account that you use to manage your GameLift game servers.
   - AWS account that you use to manage your non-GameLift resources.

   If you’re using the same account for GameLift and non-GameLift resources, you need ID and credentials for that account only.

2. **Get identifiers for each VPC.**

   Get the following information for the two VPCs to be peered:
Set up VPC Peering for an Existing Fleet

- VPC for your GameLift game servers – This is your GameLift fleet ID. Your game servers are deployed in GameLift on a fleet of EC2 instances. A fleet is automatically placed in its own VPC, which is managed by the GameLift service. You don’t have direct access to the VPC, so it is identified by the fleet ID.
- VPC for your non-GameLift AWS resources – You can establish a VPC peering with any resources that run on AWS and are managed by an AWS account that you have access to. If you haven’t already created a VPC for these resources, see Getting Started with Amazon VPC. Once you have created a VPC, you can find the VPC ID by signing into the AWS Management Console for Amazon VPC and viewing your VPCs.

Note
When setting up a peering, both VPCs must exist in the same region. The VPC for your GameLift fleet game servers is in the same region as the fleet.

3. Authorize a VPC peering with non-GameLift resources.

In this step, you are pre-authorizing a future request from GameLift to peer with your VPC for non-GameLift resources. This step updates the security group for your VPC.

To authorize the VPC peering, call the GameLift service API CreateVpcPeeringAuthorization() or use the AWS CLI command create-vpc-peering-authorization. Make this call using the account that manages your non-GameLift resources. Identify the following information:

- Peer VPC ID – This is for the VPC with your non-GameLift resources.
- GameLift AWS account ID – This is the account that you use to manage your GameLift fleet.

Once you’ve authorized a VPC peering, the authorization remains valid for 24 hours unless revoked. You can manage your VPC peering authorizations using the following operations:

- DescribeVpcPeeringAuthorizations() (AWS CLI describe-vpc-peering-authorizations).

4. Request a peering between VPCs for a GameLift fleet and the non-GameLift resources.

With a valid authorization, you can trigger GameLift to request the peering.

To request a VPC peering, call the GameLift service API CreateVpcPeeringConnection() or use the AWS CLI command create-vpc-peering-connection. Make this call using the account that manages your GameLift game servers. Identify the following information, which identifies the two VPCs to peer:

- Peer VPC ID and AWS account ID – This is the VPC for your non-GameLift resources and the account that you use to manage them. The VPC ID used must match one on a valid authorization.
- Fleet ID – This identifies the VPC for your GameLift game servers.

You can manage your VPC peering connections using the following operations:

- DescribeVpcPeeringConnections() (AWS CLI describe-vpc-peering-connections).
- DeleteVpcPeeringConnection() (AWS CLI delete-vpc-peering-connection).

5. Track your VPC peering connections.

Requesting a VPC peering connection is an asynchronous operation. To track the status of a peering request and handle success or failure cases, use one of the following options:

- Continuously poll with DescribeVpcPeeringConnections(). This operation retrieves the VPC peering connection record, including the status of the request. If a peering connection is
successfully created, the connection record also contains a CIDR block of private IP addresses that is assigned to the VPC.

- Handle fleet events associated with VPC peering connections with `DescribeFleetEvents()`, including success and failure events.

Common reasons a connection request fails include:

- An authorization for the requested connection was not found. This may mean an existing authorization is no longer valid. A common cause for this issue is a region mix-up. Verify that your authorization and your request are using the same region.

- Overlapping CIDR blocks (see Invalid VPC Peering Connection Configurations). The IPv4 CIDR blocks that are assigned to peered VPCs cannot overlap. The CIDR block of the VPC for your GameLift fleet is automatically assigned and can’t be changed. You can look up this CIDR block by calling `DescribeVpcPeeringConnections()`. To resolve this issue, you’ll need to change the CIDR block of the VPC for your non-GameLift resources to a non-overlapping range.

- The fleet did not activate. If you requested a VPC peering connection as part of a `CreateFleet()` request, the new fleet may have failed to progress to Active status. In this scenario, the peering connection cannot succeed.

Create a New Fleet with VPC Peering

You can create a new Amazon GameLift fleet and request a VPC peering connection at the same time. You can establish VPC peering with any resources that run on AWS and are managed by an AWS account that you have access to.

If you haven’t already set up a VPC for your non-GameLift resources,

- Call the GameLift command `create-vpc-peering-authorization` to pre-authorize the peering request. You’ll need the ID of the account that your use with GameLift. This authorization remains valid for 24 hours unless revoked.

To create a VPC peering connection with a new fleet:

1. **Get AWS Account ID(s) and credentials.**
   - You need an ID and sign-in credentials for the following two AWS accounts. You can find AWS account IDs by signing into the AWS Management Console and viewing your account settings. To get credentials, go to the IAM console.
     - AWS account that you use to manage your GameLift game servers.
     - AWS account that you use to manage your non-GameLift resources.

   If you’re using the same account for GameLift and non-GameLift resources, you need ID and credentials for that account only.

2. **Get the VPC ID for your non-GameLift AWS resources.**
   - If you haven’t already created a VPC for these resources, do so now (see Getting Started with Amazon VPC). Be sure that the VPC is in the same region as where you plan to create your new fleet. When creating the VPC, use the credentials for the account that manages your non-GameLift resources.

   Once you have created a VPC, you can find the VPC ID by signing into the AWS Management Console for Amazon VPC and viewing your VPCs.
3. **Authorize a VPC peering with non-GameLift resources.**

   When GameLift creates the new fleet and VPC, it also sends a request to peer with the VPC for your non-GameLift resources. You need to pre-authorize that request. This step updates the security group for your VPC.

   Using the account that manages your non-GameLift resources, call the GameLift service API `CreateVpcPeeringAuthorization()` or use the AWS CLI command `create-vpc-peering-authorization`. Identify the following information:

   - Peer VPC ID – ID of the VPC with your non-GameLift resources.
   - GameLift AWS account ID – ID for the account that you use to manage your GameLift fleet.

   Once you’ve authorized a VPC peering, the authorization remains valid for 24 hours unless revoked. You can manage your VPC peering authorizations using the following operations:

     - `DescribeVpcPeeringAuthorizations()` (AWS CLI `describe-vpc-peering-authorizations`).

4. Follow the instructions for creating a new fleet using the AWS CLI (p. 104). Include the following additional parameters:

   - `peer-vpc-aws-account-id` – ID for the account that you use to manage the VPC with your non-GameLift resources.
   - `peer-vpc-id` – ID of the VPC with your non-GameLift account.

A successful call to `create-fleet` with the VPC peering parameters generates both a new fleet and a new VPC peering request. The fleet’s status is set to `New` and the fleet activation process is initiated. The peering request’s status is set to `initiating-request`. You can track the success or failure of the peering request by calling `describe-vpc-peering-connections`.

When requesting a VPC peering connection with a new fleet, both actions either succeed or fail. If a fleet fails during the creation process, the VPC peering connection will not be established. Likewise, if a VPC peering connection fails for any reason, the new fleet will fail to move from status `Activating` to `Active`.

**Note**

The new VPC peering connection is not completed until the fleet is ready to become active. As a result, it is not available during installation of the game server build on a new fleet instance.

The following example shows a request to create both a new fleet and a peering connection between a pre-established VPC and the VPC that is created for the new fleet. The pre-established VPC is uniquely identified by the combination of your non-GameLift AWS account ID and the VPC ID.

```
$ aws gamelift create-fleet
  --name "My_Fleet_1"
  --description "The sample test fleet"
  --ec2-instance-type "c5.large"
  --fleet-type "ON_DEMAND"
  --build-id "build-1111aaaa-22bb-33cc-44dd-5555eeee66ff"
  --runtime-configuration "GameSessionActivationTimeoutSeconds=300,
    MaxConcurrentGameSessionActivations=2,
    ServerProcesses=[{LaunchPath=C:\game\Bin64.dedicated
      MultiplayerSampleProjectLauncher_Server.exe,
      Parameters=+sv_port 33435 +start_lobby,
      ConcurrentExecutions=10}]
  --new-game-session-protection-policy "FullProtection"
  --resource-creation-limit-policy "NewGameSessionsPerCreator=3,
    PolicyPeriodInMinutes=15"
  --ec2-inbound-permissions "FromPort=33435,ToPort=33435,IpRange=0.0.0.0/0,Protocol=UDP"
  "FromPort=33235,ToPort=33235,IpRange=0.0.0.0/0,Protocol=UDP"
```
--MetricGroups "EMEAfleets"
--peer-vpc-aws-account-id "111122223333"
--peer-vpc-id "vpc-allall1a"

Copyable version:

```
aws gamelift create-fleet --name "My_Fleet_1" --description "The sample test fleet" --fleet-type "ON_DEMAND" --MetricGroups "EMEAfleets" --build-id "build-1111aaaa-22bb-33cc-44dd-5555eee66ff" --ec2-instance-type "c5.large" --runtime-configuration "GameSessionActivationTimeoutSeconds=300,MaxConcurrentGameSessionActivations=2,ServerProcesses=[{LaunchPath=C:\game\Bin64.dedicated\MultiplayerSampleProjectLauncher_Server.exe,Parameters=+/sv_port 33435 +start_lobby,ConcurrentExecutions=10}]" --new-game-session-protection-policy "FullProtection" --resource-creation-limit-policy "NewGameSessionsPerCreator=3,PolicyPeriodInMinutes=15" --ec2-inbound-permissions "FromPort=33435,ToFPort=33435,IpRange=0.0.0.0/0,Protocol=UDP" "FromPort=33235,ToFPort=33235,IpRange=0.0.0.0/0,Protocol=UDP" --peer-vpc-aws-account-id "111122223333" --peer-vpc-id "vpc-allall1a"
```
Viewing Your Game Data in the Console

The managed GameLift service continually collects data for active games to help you understand player behavior and performance. With the Amazon GameLift console, you can view, manage, and analyze this information for your builds, fleets, game sessions, and player sessions.

Topics
- View Your Current Amazon GameLift Status (p. 170)
- View Your Builds (p. 171)
- View Your Fleets (p. 172)
- View Fleet Details (p. 172)
- View Data on Game and Player Sessions (p. 176)
- View Your Aliases (p. 177)

View Your Current Amazon GameLift Status

The Dashboard provides a grid view showing the following:

- Builds
- Fleets in all statuses
- Aliases and the fleets they point to (if any)

To open the Amazon GameLift dashboard
- In the Amazon GameLift console, choose Dashboard from the menu bar.

From the dashboard you can take the following actions:

- View relationships among items. Click anywhere inside an item box to show the relationships between that item and others on the dashboard. For example, click a build to display all fleets that were created with that build. Click a fleet to see the build it was created with and the alias it points to. To reset the dashboard, click the Reset overview button.
- View details on a build, fleet, or alias. Click the ID number for a item to open the details page.
View Your Builds

You can view information about all the game server builds you have uploaded to Amazon GameLift and take actions on them. Builds shown include only those uploaded for the selected region.

Build Catalog

Uploaded builds are shown on the Builds page. To view this page, choose Builds from the Amazon GameLift console menu bar.

The Builds page provides the following summary information for all builds:

- **Status** – Displays one of three possible status messages:
  - **Initialized** – The build has been created, but the upload has not yet started or the upload is still in progress.
  - **Ready** – The build has been successfully received and is ready for fleet creation.
  - **Failed** – The build timed out before the binaries were received.
- **Build name** – Name associated with the uploaded build. A build name is provided when uploading the build to Amazon GameLift, and can be changed using the AWS SDK action UpdateBuild.
- **Build ID** – Unique ID assigned to the build on upload.
- **Version** – Version label associated with the uploaded build. A build name is provided when uploading the build to Amazon GameLift, and can be changed using the AWS SDK action UpdateBuild.
- **OS** – Operating system that the build runs on. The build OS determines what operating system is installed on a fleet's instances.
- **Size** – Size, in megabytes (MB) of the build file uploaded to Amazon GameLift.
- **Date created** – Date and time that the build was uploaded to Amazon GameLift.
- **Fleets** – Number of fleets currently deployed with this build.

From this page you can do any of the following:

- Create a new fleet from a build. Select a build and click Create fleet from build.
- Delete a build. Select a build and click Delete build.
• Filter and sort the build list. Use the controls at the top of the table.
• View build details. Click a build name to open the build detail page.

Build Detail

Access a build's detail page from either the console dashboard or the Builds page by clicking the build name. The Build detail page displays the same build summary information as the Builds page. It also shows a list of fleets created with the build. This list is essentially the fleets catalog, filtered by build. It includes the same summary information as the Fleets page (p. 172).

View Your Fleets

You can view information on all the fleets created to host your games on Amazon GameLift under your AWS account. Fleets shown include only those created in the selected region. From the Fleets page, you can create a new fleet or view additional detail on a selected fleet. A fleet's detail page (p. 172) contains usage information and metrics as well as game session and player session data; you can also edit the fleet record or terminate the fleet.

To view the Fleets page, choose Fleets from the Amazon GameLift console's menu bar.

The Fleets page displays the following summary information by default. You can customize which information to display by using the Settings (gear) button.

• Status – The status of the fleet, which can be one of these states: New, Downloading, Building, and Active. A fleet must be in Active status to be able to host game sessions and allow player connections.
• Fleet name – Friendly name given to the fleet.
• EC2 type – The Amazon EC2 instance type, which determines the computing capacity of fleet's instances.
• OS – Operating system on each instances in the fleet. A fleet's OS is determined by the build deployed to it.
• Active instances – The number of EC2 instances in use for the fleet.
• Maximum instances – The current maximum number of EC2 instances allowed on the fleet. This value is configurable (within service limits) from the Fleet detail page, Scaling tab.
• Game sessions – The number of active game sessions currently running in the fleet. The data is delayed five minutes.
• Player sessions – The number of players connected to game sessions in the fleet. The data is delayed five minutes.
• Uptime – The total length of time the fleet has been running.
• Date created – The date and time the fleet was created.

View Fleet Details

You can access detailed information on any fleet, including configuration settings, scaling settings, metrics, and game and player data. Access a Fleet detail page from either the console dashboard or the Fleets page by clicking the fleet name.

On this page you can also take the following actions:

• Update the fleet's metadata and runtime configuration. Choose Actions: Edit fleet.
• Change fleet capacity settings. On the Scaling page, edit the Minimum and Maximum instance counts, or manually set the Desired fleet capacity to a fixed size.
• Set or change auto-scaling policies. On the Scaling page, enable or disable auto-scaling.
• Shut down a fleet. Choose Actions: Terminate fleet.

Fleet Summary

The summary table includes the following information:

• Status – Current status of the fleet, which may be New, Downloading, Building, and Active. A fleet must be active before it can host game sessions or accept player connections.
• Fleet ID – Unique identifier assigned to the fleet.
• OS – Operating system on each instances in the fleet. A fleet's OS is determined by the build deployed to it.
• Fleet type – Indicates whether the fleet uses on-demand or spot instances.
• EC2 type – Amazon EC2 instance type selected for the fleet when it was created. A fleet's instance type specifies the computing hardware and capacity used for each instance in the fleet and determines the instance limits for the fleet.
• Active instances – Number of instances in Active status, which means they are currently running game sessions or are ready to run game sessions.
• Active servers – Number of server processes currently in an Active status in the fleet. The data has a five-minute delay.
• Game sessions – Number of active game sessions running on instances in the fleet. The data has a five-minute delay.
• Player sessions – Number of players connected along with the total number of player slots in active game sessions across the fleet. For example: 25 (connected players) of 100 (possible players) means the fleet can support 75 additional players. The data has a five-minute delay.
• Protection – Current setting for game session protection (p. 6) for the fleet.
• Uptime – Total length of time the fleet has been active.
• Date created – Date and time indicating when the fleet was created.

Metrics

The Metrics tab displays a graphical representation of fleet metrics over time. To view metrics using Amazon CloudWatch, see Monitor Amazon GameLift with Amazon CloudWatch (p. 178).

To display metrics information in the graph

1. From the list shown to the left of the graph area, click a metric name to add it to the graph display. Metric names that are gray are currently not being graphed. The color key identifies which line matches each graphed metric. Descriptions of individual metrics can be found at Amazon GameLift Metrics for Fleets (p. 178). The following categories of metrics are available:

   • Instance Counts – These metrics track changes in capacity and utilization at the instance level over time (also shown on the Scaling tab).
   • Game – These metrics show fleet activity and utilization at the game session level over time.
   • Server processes – These metrics track the status and health of server processes across the fleet. The Amazon GameLift service regularly polls each active server process for its health.
   • Instance performance – These metrics reflect performance of the fleet's computing resources. See detailed descriptions of each metric at EC2 Instance Metrics.
2. Use the following filters, shown above the graph area, to change how metric data is displayed:

   - **Data & Period** – Offers two options for selecting a date range:
     - Use **Relative** to select a period of time relative to the current time, such as *Last hour, Last day, Last week*.
     - Use **Absolute** to specify a period with an arbitrary start and end date/time.
   - **Granularity** – Select a length of time to aggregate data points.
   - **Refresh rate** – Select how often you want the graph display to be updated.
   - **Time zone** – Select which time format to use in the graph display: **UTC** (universal coordinated time) or **Browser time** (local time).
   - **Show points** – Toggle on or off to display discrete data points (as circles) or display lines only.

---

**Events**

The **Events** tab provides a log of all events that have occurred on the fleet, including the event code, message, and time stamp. See Event descriptions in the Amazon GameLift API Reference.

**Scaling**

The **Scaling** tab contains information related to fleet capacity, including the current status and a graphical representation of capacity changes over time. It also provides tools to update capacity limits and manage auto-scaling.

**To view current and historical scaling information**

1. Go to the top of the **Scaling** tab. The current capacity status for this fleet is shown in the left column. These values are defined as follows:

   - **Scaling Limits** – These metrics track the history of changes to capacity limits.
     - **Minimum** – Hard lower limit on the number of instances to maintain in the fleet. Fleet capacity will not drop below the current minimum during auto-scaling or even if desired capacity is set below the minimum.
     - **Desired** – The number of active instances currently **wanted** in the fleet. The goal is to make the number of **Active** instances (explained later) match the number of desired instances; it achieves this by creating or terminating instances as needed.
     - **Maximum** – Hard upper limit on the number of instances to maintain in the fleet. Fleet capacity will not exceed the current maximum during auto-scaling or if desired capacity is set above the maximum.
   - **Instance Counts** and **Game** – These metrics track changes in capacity and utilization over time. See descriptions of individual metrics at Amazon GameLift Metrics for Fleets (p. 178).

   To view how fleet capacity has changed over time, display in the graph any or all of the scaling metrics listed on the left. Click the metric name to add it to the graph. (Metric names are gray when not in use.) Use the color key to identify which graphed line matches a selected metric.

2. Use the following filters, shown above the graph area, to specify how metric data is displayed in the graph:

   - **Data & Period** – Offers two options for selecting a date range:
     - Use **Relative** to select a period of time relative to the current time, such as *Last hour, Last day, Last week*.
     - Use **Absolute** to specify a period with an arbitrary start and end date/time.
   - **Granularity** – Select a length of time to aggregate data points.
• **Refresh rate** – Select how often you want the graph display to be updated.
• **Format** – Select which time format to use in the graph display: **UTC** (universal coordinated time) or **Browser time** (local time).
• **Show Points** – Toggle on or off to display discrete data points (as circles) or display lines only.

**To change fleet capacity**

- Below the history graph are tools for setting capacity limits and scaling capacity. See [Manually Set Fleet Capacity](#) for details on how to use these tools. See [Auto-Scale Fleet Capacity](#) for information about managing auto-scaling.

**Game sessions**

The **Game sessions** tab lists past and present game sessions hosted on the fleet, including some detail information. Click a game session ID to access additional game session information, including player sessions. See [View Data on Game and Player Sessions](#) for more details.

**Capacity allocation**

The **Capacity allocation** tab displays the runtime configuration for the fleet, which specifies what server processes to launch on each instance. It includes the path for the game server executable and optional launch parameters. You can change the fleet's capacity allocation either by editing the fleet in the console or by using the AWS CLI to update the runtime configuration.

**Ports**

The **Ports** tab displays the fleet's connection permissions, including IP address and port setting ranges. You can change connection permissions by either editing the fleet in the console or using the AWS CLI to update the fleet's port settings.

**Build**

The **Build** tab displays the fleet's build-related configuration, which was set when the fleet was created. Select the build ID to see the full **build** detail page.

If your build has been deleted or an error has occurred while attempting to retrieve your build, you may see one of the following status messages:

- **Deleted** – The build for this fleet was deleted. Your fleet will still run properly despite the build having been deleted.
- **Error** – An error occurred while attempting to retrieve build information for the fleet.

**ARNs**

The **ARNs** tab lists **Amazon Resource Name (ARN) associated with this fleet:**

- **Fleet ARN** – The identifier assigned to this fleet. A fleet's ARN identifies it as an Amazon GameLift resource and specifies the region and AWS account, to ensure that it is a unique identifier.
- **Instance Role ARN** – An identifier for an AWS IAM role that manages access to your other AWS resources, if one was provided during fleet creation. When a role ARN is associated with the fleet, the game servers and other applications that are running on the fleet are able to access those other AWS resources. Learn more at [Access AWS Resources From Your Fleets](#).
View Data on Game and Player Sessions

You can view information about the games running on a fleet and as well as individual players. For more information about game sessions and player sessions, see How Players Connect to Games (p. 12).

To view game session data

1. In the Amazon GameLift console, open the detail page for the fleet you want to study. (Choose Fleets in the menu bar and click on a fleet name.)
2. Open the Game sessions tab. This tab lists all game sessions hosted on the fleet along with summary information.
3. Click a game session to view additional information about the game session as well as a list of players that were connected to the game.

Game sessions

A summary of your game session information is displayed at the top of the page and includes:

- **Status** – Game session status. Valid statuses are:
  - *Activating* – A game session has been initiated and is preparing to run.
  - *Active* – A game session is running and available to receive players (depending on the session's player creation policy).
  - *Terminated* – Game session has ended.
- **Name** – Game generated for the game session.
- **ID** – Unique identifier assigned by Amazon GameLift to the game session.
- **IP address** – IP address specified for the game session.
- **Port** – Port number used to connect to the game session.
- **Player sessions** – Number of players connected to the game sessions along with total possible players the game session can support. For example: 2 (connected players) of 10 (possible players) means the fleet can support 8 additional players.
- **Uptime** – Total length of time the game session has been running.
- **Date created** – Date and time stamp indicating when the fleet was created.

Player sessions

The following player session data is collected for each game session:

- **Status** – The status of the player session. Options include:
  - *Reserved* – Player session has been reserved, but the player has not yet connected.
  - *Active* – Player session is currently connected to the game server.
  - *Completed* – Player session has ended; player is no longer connected.
  - *Timed Out* – Player session was reserved, but the player failed to connect.
- **ID** – The identifier assigned to the player session.
- **Player ID** – A unique identifier for the player. Click this ID to get additional player information.
- **Start time** – The time the player connected to the game session.
- **End time** – The time the player disconnected from the game session.
- **Total time** – The total length of time the player has been active in the player session.
Player information

View additional information for a selected player, including a list of all games the player connected to across all fleets in the current region. This information includes the status, start and end times, and total connected time for each player session. You can click to view data for the relevant game sessions and fleets.

View Your Aliases

You can view information on all of the fleet aliases you have created and take actions on them on the Aliases page. Aliases shown include only those created for the selected region.

Alias Catalog

All created aliases are shown on the Aliases catalog page. To view the Aliases page, choose Aliases from the Amazon GameLift console's menu bar.

The Aliases page provides summary information on all builds, including type. From this page you can:

- Create a new alias. Click Create alias.
- Filter and sort the aliases list. Use the controls at the top of the table.
- View alias details. Click an alias name to open the alias detail page.

Alias Detail

Access an alias's detail page from either the console dashboard or the Aliases catalog page by clicking the alias name. The Alias detail page displays a summary of information on the alias.

From this page you can:

- Edit an alias, including changing the name, description, and the fleet ID the alias is associated with. Click Actions: Edit alias.
- View information on the fleet the alias is currently associated with. This includes the fleet's status and current utilization (active game sessions and players).
- Delete an alias. Click Actions: Delete alias.

Alias detail information includes:

- **Type** – The routing option for the alias, which can be one of these:
  - **Simple** – A simple alias routes a player to games on an associated fleet. You can update the alias to point to a different fleet at any time.
  - **Terminal** – A terminal alias does not point to a fleet. Instead it passes a message back to the client. This alias type is useful for gracefully notifying players when a set of game servers is no longer available. For example, a terminal alias might inform players that their game clients are out of date and provide upgrade options.
- **Alias ID** – The unique number used to identify the alias.
- **Description** – The description of the alias.
- **Date created** – The date and time the alias was created.
Monitoring Amazon GameLift

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon GameLift and your other AWS solutions. There are three primary uses for metrics with Amazon GameLift: to monitor system health and set up alarms, to track game server performance and usage, and to manage capacity using manual or auto-scaling.

AWS provides the following monitoring tools to watch Amazon GameLift, report when something is wrong, and take automatic actions when appropriate:

- Amazon GameLift Console
- Amazon CloudWatch — You can monitor Amazon GameLift metrics in real time, as well as metrics for other AWS resources and applications that you're running on AWS services. CloudWatch offers a suite of monitoring features, including tools to create customized dashboards and the ability to set alarms that notify or take action when a metric reaches a specified threshold.
- AWS CloudTrail — captures all API calls and related events made by or on behalf of your AWS account for Amazon GameLift and other AWS services. Data is delivered as log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred.

Topics
- Monitor Amazon GameLift with Amazon CloudWatch (p. 178)
- Logging Amazon GameLift API Calls with AWS CloudTrail (p. 188)

Monitor Amazon GameLift with Amazon CloudWatch

You can monitor Amazon GameLift using Amazon CloudWatch, an AWS service that collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so that you can access historical information and gain a better perspective on how your game server hosting with Amazon GameLift is performing. You can also set alarms that watch for certain thresholds and send notifications or take actions when those thresholds are met. For more information, see the Amazon CloudWatch User Guide.

To access CloudWatch metrics for your games, you can use the AWS Management Console, the AWS CLI, or the CloudWatch API. The following tables list the metrics and dimensions for Amazon GameLift. All metrics that are available in CloudWatch are also available in the Amazon GameLift console, which provides the data as a set of customizable graphs.

Amazon GameLift Metrics for Fleets

The AWS/GameLift namespace includes the following metrics related to activity across a fleet or a group of fleets. The Amazon GameLift service sends metrics to CloudWatch every minute.

Instances

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveInstances</td>
<td>Instances with ACTIVE status, which means they are running active server processes. The count includes idle instances and those that are hosting one or more games</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>sessions</strong></td>
<td>This metric measures current total instance capacity. This metric can be used with automatic scaling.</td>
</tr>
<tr>
<td>Units: Count</td>
<td></td>
</tr>
<tr>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
<td></td>
</tr>
<tr>
<td><strong>DesiredInstances</strong></td>
<td>Target number of active instances that Amazon GameLift is working to maintain in the fleet. With automatic scaling, this value is determined based on the scaling policies currently in force. Without automatic scaling, this value is set manually. This metric is not available when viewing data for fleet metric groups.</td>
</tr>
<tr>
<td>Units: Count</td>
<td></td>
</tr>
<tr>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
<td></td>
</tr>
<tr>
<td><strong>IdleInstances</strong></td>
<td>Active instances that are currently hosting zero (0) game sessions. This metric measures capacity that is available but unused. This metric can be used with automatic scaling.</td>
</tr>
<tr>
<td>Units: Count</td>
<td></td>
</tr>
<tr>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
<td></td>
</tr>
<tr>
<td><strong>MaxInstances</strong></td>
<td>Maximum number of instances that are allowed for the fleet. A fleet's instance maximum determines the capacity ceiling during manual or automatic scaling up. This metric is not available when viewing data for fleet metric groups.</td>
</tr>
<tr>
<td>Units: Count</td>
<td></td>
</tr>
<tr>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
<td></td>
</tr>
<tr>
<td><strong>MinInstances</strong></td>
<td>Minimum number of instances allowed for the fleet. A fleet's instance minimum determines the capacity floor during manual or automatic scaling down. This metric is not available when viewing data for fleet metric groups.</td>
</tr>
<tr>
<td>Units: Count</td>
<td></td>
</tr>
<tr>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
<td></td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PercentIdleInstances</td>
<td>Percentage of all active instances that are idle (calculated as IdleInstances / ActiveInstances). This metric can be used for automatic scaling.</td>
</tr>
<tr>
<td></td>
<td>Units: Percent</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
</tr>
<tr>
<td>InstanceInterruptions</td>
<td>Number of spot instances that have been interrupted.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum, Average, Minimum, Maximum</td>
</tr>
</tbody>
</table>

**Server Processes**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveServerProcesses</td>
<td>Server processes with ACTIVE status, which means they are running and able to host game sessions. The count includes idle server processes and those that are hosting game sessions. This metric measures current total server process capacity.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
</tr>
<tr>
<td>HealthyServerProcesses</td>
<td>Active server processes that are reporting healthy. This metric is useful for tracking the overall health of the fleet’s game servers.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
</tr>
<tr>
<td>PercentHealthyServerProcesses</td>
<td>Percentage of all active server processes that are reporting healthy (calculated as HealthyServerProcesses / ActiveServerProcesses).</td>
</tr>
<tr>
<td></td>
<td>Units: Percent</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
</tr>
<tr>
<td>ServerProcessAbnormalTerminations</td>
<td>Server processes that were shut down due to abnormal circumstances since the last report. This metric includes terminations that were initiated by the Amazon GameLift service. This occurs when a server process</td>
</tr>
</tbody>
</table>
## Amazon GameLift Metrics for Fleets

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Metric</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>ServerProcessActivations</strong></td>
<td>Server processes that successfully transitioned from ACTIVATING to ACTIVE status since the last report. Server processes cannot host game sessions until they are active.</td>
</tr>
<tr>
<td><strong>Units:</strong> Count</td>
<td>Sum, Average, Minimum, Maximum</td>
</tr>
<tr>
<td><strong>ServerProcessTerminations</strong></td>
<td>Server processes that were shut down since the last report. This includes all server processes that transitioned to TERMINATED status for any reason, including normal and abnormal process terminations.</td>
</tr>
<tr>
<td><strong>Units:</strong> Count</td>
<td>Sum, Average, Minimum, Maximum</td>
</tr>
</tbody>
</table>

## Game Sessions

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Metric</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>ActivatingGameSessions</strong></td>
<td>Game sessions with ACTIVATING status, which means they are in the process of starting up. Game sessions cannot host players until they are active. High numbers for a sustained period of time may indicate that game sessions are not transitioning from ACTIVATING to ACTIVE status. This metric can be used with automatic scaling.</td>
</tr>
<tr>
<td><strong>Units:</strong> Count</td>
<td>Average, Minimum, Maximum</td>
</tr>
<tr>
<td><strong>ActiveGameSessions</strong></td>
<td>Game sessions with ACTIVE status, which means they are able to host players, and are hosting zero or more players. This metric measures the total number of game sessions currently being hosted. This metric can be used with automatic scaling.</td>
</tr>
<tr>
<td><strong>Units:</strong> Count</td>
<td></td>
</tr>
</tbody>
</table>
## Amazon GameLift Metrics for Fleets

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AvailableGameSessions</strong></td>
<td>Game session slots on active, healthy server processes that are not currently being used. This metric measures the number of new game sessions that could be started immediately. This metric can be used with automatic scaling.</td>
</tr>
<tr>
<td><strong>Units</strong>: Count</td>
<td></td>
</tr>
<tr>
<td><strong>Relevant CloudWatch statistics</strong>: Average, Minimum, Maximum</td>
<td></td>
</tr>
<tr>
<td><strong>PercentAvailableGameSessions</strong></td>
<td>Percentage of game session slots on all active server processes (healthy or unhealthy) that are not currently being used (calculated as AvailableGameSessions / [ActiveGameSessions + AvailableGameSessions + unhealthy server processes]). This metric can be used with automatic scaling.</td>
</tr>
<tr>
<td><strong>Units</strong>: Percent</td>
<td></td>
</tr>
<tr>
<td><strong>Relevant CloudWatch statistics</strong>: Average</td>
<td></td>
</tr>
<tr>
<td><strong>GameSessionInterruptions</strong></td>
<td>Number of game sessions on spot instances that have been interrupted.</td>
</tr>
<tr>
<td><strong>Units</strong>: Count</td>
<td></td>
</tr>
<tr>
<td><strong>Relevant CloudWatch statistics</strong>: Sum, Average, Minimum, Maximum</td>
<td></td>
</tr>
</tbody>
</table>

## Player Sessions

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CurrentPlayerSessions</strong></td>
<td>Player sessions with either ACTIVE status (player is connected to an active game session) or RESERVED status (player has been given a slot in a game session but hasn't yet connected). This metric can be used with automatic scaling.</td>
</tr>
<tr>
<td><strong>Units</strong>: Count</td>
<td></td>
</tr>
<tr>
<td><strong>Relevant CloudWatch statistics</strong>: Average, Minimum, Maximum</td>
<td></td>
</tr>
<tr>
<td><strong>PlayerSessionActivations</strong></td>
<td>Player sessions that transitioned from RESERVED status to ACTIVE since the last report. This occurs when a player successfully connects to an active game session.</td>
</tr>
<tr>
<td><strong>Units</strong>: Count</td>
<td></td>
</tr>
</tbody>
</table>
Amazon GameLift Metrics for Queues

The GameLift namespace includes the following metrics related to activity across a game session placement queue. The Amazon GameLift service sends metrics to CloudWatch every minute.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AverageWaitTime</strong></td>
<td>Average amount of time that game session placement requests in the queue with status PENDING have been waiting to be fulfilled. Units: Seconds Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
</tr>
<tr>
<td><strong>FirstChoiceNotViable</strong></td>
<td>Game sessions that were successfully placed but NOT in the first-choice fleet, because that fleet was considered not viable (such as a spot fleet with a high interruption rate). The first-choice fleet is either the first fleet listed in the queue or—when a placement request includes player latency data—it is the first fleet chosen by FleetIQ prioritization. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td><strong>FirstChoiceOutOfCapacity</strong></td>
<td>Game sessions that were successfully placed but NOT in the first-choice fleet, because that fleet had no available resources. The first-choice fleet is either the first fleet listed in the queue or—when a placement request includes player latency data —it is the first fleet chosen by FleetIQ prioritization. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td><strong>LowestLatencyPlacement</strong></td>
<td>Game sessions that were successfully placed in a region that offers the queue's lowest possible latency for the players. This metric is emitted only when player latency data is included in the placement request, which triggers FleetIQ prioritization. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LowestPricePlacement</td>
<td>Game sessions that were successfully placed in a fleet with the queue's lowest possible price for the chosen region. (FleetIQ prioritization first chooses the region with the lowest latency for the players and then finds the lowest cost fleet within that region.) This fleet can be either a spot fleet or an on-demand instance if the queue has no spot instances. This metric is emitted only when player latency data is included in the placement request. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td>Placement &lt;region name&gt;</td>
<td>Game sessions that are successfully placed in fleets located in the specified region. This metric breaks down the PlacementsSucceeded metric by region. Units: Count Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td>PlacementsCanceled</td>
<td>Game session placement requests that were canceled before timing out since the last report. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td>PlacementsFailed</td>
<td>Game session placement requests that failed for any reason since the last report. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td>PlacementsStarted</td>
<td>New game session placement requests that were added to the queue since the last report. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td>PlacementsSucceeded</td>
<td>Game session placement requests that resulted in a new game session since the last report. Units: Count Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
</tbody>
</table>
### Amazon GameLift Metrics for Matchmaking

The GameLift namespace includes metrics on matchmaking activity for matchmaking configurations and matchmaking rules. The Amazon GameLift service sends metrics to CloudWatch every minute.

For more information on the sequence of matchmaking activity, see [How Amazon GameLift FlexMatch Works](#).

#### Matchmaking Configurations

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CurrentTickets</strong></td>
<td>Matchmaking requests currently being processed or waiting to be processed.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum, Sum</td>
</tr>
<tr>
<td><strong>MatchAcceptancesTimedOut</strong></td>
<td>For matchmaking configurations that require acceptance, the potential matches that timed out during acceptance since the last report.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td><strong>MatchesAccepted</strong></td>
<td>For matchmaking configurations that require acceptance, the potential matches that were accepted since the last report.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td><strong>MatchesCreated</strong></td>
<td>Potential matches that were created since the last report.</td>
</tr>
</tbody>
</table>

---

Version 185
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatchesPlaced</td>
<td>Matches that were successfully placed into a game session since the last report.</td>
</tr>
<tr>
<td>Units: Count</td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td>MatchesRejected</td>
<td>For matchmaking configurations that require acceptance, the potential matches that were rejected by at least one player since the last report.</td>
</tr>
<tr>
<td>Units: Count</td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td>PlayersStarted</td>
<td>Players in matchmaking tickets that were added since the last report.</td>
</tr>
<tr>
<td>Units: Count</td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td>TicketsFailed</td>
<td>Matchmaking requests that resulted in failure since the last report.</td>
</tr>
<tr>
<td>Units: Count</td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td>TicketsStarted</td>
<td>New matchmaking requests that were created since the last report.</td>
</tr>
<tr>
<td>Units: Count</td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td>TicketsTimedOut</td>
<td>Matchmaking requests that reached the timeout limit since the last report.</td>
</tr>
<tr>
<td>Units: Count</td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td>TimeToMatch</td>
<td>For matchmaking requests that were put into a potential match before the last report, the amount of time between ticket creation and potential match creation.</td>
</tr>
<tr>
<td>Units: Seconds</td>
<td>Relevant CloudWatch statistics: Data Samples, Average, Minimum, Maximum, p99</td>
</tr>
</tbody>
</table>
### Matchmaking Rules

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RuleEvaluationsPassed</strong></td>
<td>Rule evaluations during the matchmaking process that passed since the last report. This metric is limited to the top 50 rules.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td><strong>RuleEvaluationsFailed</strong></td>
<td>Rule evaluations during matchmaking that failed since the last report. This metric is limited to the top 50 rules.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
</tbody>
</table>

### Dimensions for Amazon GameLift Metrics

Amazon GameLift supports filtering metrics by the following dimensions.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FleetId</strong></td>
<td>Unique identifier for a single fleet. This dimension is used with all metrics for instances, server processes, game sessions, and player sessions. It is not used with metrics for queues and matchmaking.</td>
</tr>
<tr>
<td><strong>MetricGroup</strong></td>
<td>Unique identifier for a collection of fleets. A fleet is included in a fleet metric group by adding the metric group name to the fleet’s attributes (see UpdateFleetAttributes()). This dimension is used with all</td>
</tr>
<tr>
<td>Dimension</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>metrics for instances, server processes, game sessions, and player sessions. It is not used with metrics for queues and matchmaking.</td>
</tr>
<tr>
<td>QueueName</td>
<td>Unique identifier for a single queue. This dimension is used with metrics for game session placement queues only.</td>
</tr>
<tr>
<td>MatchmakingConfigurationName</td>
<td>Unique identifier for a single matchmaking configuration. This dimension is used with metrics for matchmaking only.</td>
</tr>
<tr>
<td>MatchmakingConfigurationName-RuleName</td>
<td>Unique identifier for the intersect of a matchmaking configuration and a matchmaking rule. This dimension is used with metrics for matchmaking only.</td>
</tr>
<tr>
<td>InstanceType</td>
<td>Unique identifier for an EC2 instance type designation, such as &quot;c4.large&quot;. This dimension is used with metrics for spot instances only.</td>
</tr>
<tr>
<td>OperatingSystem</td>
<td>Unique identifier for the operating system of an instance. This dimension is used with metrics for spot instances only.</td>
</tr>
</tbody>
</table>

Logging Amazon GameLift API Calls with AWS CloudTrail

Amazon GameLift is integrated with AWS CloudTrail, a service that captures all of the API calls made by or on behalf of Amazon GameLift in your AWS account. CloudTrail delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from the Amazon GameLift console or from the Amazon GameLift API. Using the information collected by CloudTrail, you can determine what request was made to Amazon GameLift, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the AWS CloudTrail User Guide.

Amazon GameLift Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to Amazon GameLift actions are tracked in log files. Amazon GameLift records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

All Amazon GameLift actions are logged by CloudTrail. For example, calls to CreateGameSession, CreatePlayerSession and UpdateGameSession generate entries in the CloudTrail log files. For the complete list of actions, see the Amazon GameLift API Reference.

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with AWS account root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the userIdentity field in the CloudTrail Event Reference.

You can store your log files in your S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically.
If you want to take quick action upon log file delivery, you can choose to have CloudTrail publish Amazon Simple Notification Service (Amazon SNS) notifications when new log files are delivered. For more information, see Configuring Amazon SNS Notifications.

You can also aggregate Amazon GameLift log files from multiple AWS regions and multiple AWS accounts into a single S3 bucket. For more information, see Aggregating CloudTrail Log Files to a Single Amazon S3 Bucket.

Understanding Amazon GameLift Log File Entries

CloudTrail log files can contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows a CloudTrail log entry that demonstrates the CreateFleet and DescribeFleetAttributes actions.

```
{
  "Records": [
    {
      "eventVersion": "1.04",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-12-29T23:40:15Z",
      "eventSource": "gamelift.amazonaws.com",
      "eventName": "CreateFleet",
      "awsRegion": "us-west-2",
      "sourceIPAddress": "192.0.2.0",
      "userAgent": "[]",
      "requestParameters": {
        "buildId": "build-92b6e8af-37a2-4c10-93bd-4698ea23de8d",
        "eC2InboundPermissions": [
          {
            "ipRange": "10.24.34.0/23",
            "fromPort": 1935,
            "protocol": "TCP",
            "toPort": 1935
          }
        ],
        "logPaths": [
          "C:\\game\\serverErr.log",
          "C:\\game\\serverOut.log"
        ],
        "eC2InstanceType": "c5.large",
        "serverLaunchPath": "C:\\game\\MyServer.exe",
        "description": "Test fleet",
        "serverLaunchParameters": "-paramX=baz",
        "name": "My_Test_Server_Fleet"
      },
      "responseElements": {
        "fleetAttributes": {
          "fleetId": "fleet-0bb84136-4f69-4bb2-bfec-a9b9a7c3d52e",
          "serverLaunchPath": "C:\\game\\MyServer.exe",
          "status": "NEW"
        }
      }
    }
  ]
}
```
"logPaths": [  "C:\game\serverErr.log",  "C:\game\serverOut.log" ],  "description": "Test fleet",  "serverLaunchParameters": "-paramX=baz",  "creationTime": "Dec 29, 2015 11:40:14 PM",  "name": "My_Test_Server_Fleet",  "buildId": "build-92b6e8af-37a2-4c10-93bd-4698ea23de8d" },  ",  "requestID": "824a2a4b-ae85-11e5-a8d6-61d5c4fb25f2",  "eventID": "c8f6e01-fbf9-4c4e-a0fe-ad7dc205ce11",  "eventType": "AwsApiCall",  "recipientAccountId": "111122223333" ],  ",  {  "eventVersion": "1.04",  "userIdentity": {  "type": "IAMUser",  "principalId": "AIDACKCEVSQ6C2EXAMPLE",  "arn": "arn:aws:iam::111122223333:user/myUserName",  "accountId": "111122223333",  "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  "userName": "myUserName" },  "eventTime": "2015-12-29T23:40:15Z",  "eventName": "DescribeFleetAttributes",  "awsRegion": "us-west-2",  "sourceIPAddress": "192.0.2.0",  "userAgent": "[]",  "requestParameters": {  "fleetIds": [  "fleet-0bb84136-4f69-4bb2-bfec-a9b9a7c3d52e" ]  },  "responseElements": null,  "requestID": "82e7f0ec-ae85-11e5-a8d6-61d5c4fb25f2",  "eventID": "11daabcb-009f-49f2-8b3d-3a63c8bad86f",  "eventType": "AwsApiCall",  "recipientAccountId": "111122223333"  },  "}
Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

GameLift FleetIQ, which optimizes the use of low-cost Spot Instances for cloud-based game hosting, is now available as a standalone feature. Spot Instances use spare AWS computing capacity that is offered at steep discounts because availability fluctuates and there is the potential for interruptions. Because game hosting is particularly sensitive to interruptions, Spot Instances aren't always a viable option. GameLift FleetIQ makes Spot Instances viable for game hosting by minimizing the impact of interruptions on your game servers. With GameLift FleetIQ as a standalone feature, you can work directly with your hosting resources in Amazon EC2 and Auto Scaling while taking advantage of GameLift optimizations to deliver low-cost, resilient game hosting for your players.

GameLift FleetIQ is a good game hosting option for customers who:

- Have existing AWS deployments or want to use Amazon EC2 directly rather than through the managed GameLift service. GameLift FleetIQ works with Auto Scaling groups that you manage in your AWS account, with full access to your Amazon EC2 instances and groups. You can also integrate with other AWS services, including Amazon Elastic Container Service, Amazon Elastic Kubernetes Service, and AWS Shield Advanced.
- Have existing on-premises game hosting and want to extend capacity to the cloud. With GameLift FleetIQ, you can build a hybrid deployment system that uses existing on-premises capacity while incrementally adding AWS Cloud capacity as needed.

**GameLift FleetIQ Features**

- **Optimized Spot rebalancing.** GameLift FleetIQ periodically evaluates your instance types and replaces instances that are not considered viable due to a higher potential for game session interruptions. As your Auto Scaling group retires old instances and starts new ones, the group is continually refreshed with instance types that are currently viable for game hosting.

- **Optimal player routing.** GameLift FleetIQ APIs direct new game sessions onto instances where they are least likely to be interrupted. Game sessions are also packed onto fewer instances, which improves the Auto Scaling group's ability to scale down unneeded resources, and lower costs.

- **Automatic scaling based on player usage.** GameLift FleetIQ emits game server utilization data as CloudWatch metrics. You can use these metrics to automatically scale your available hosting resources to track with actual player demand and reduce hosting costs.

- **Direct management of Amazon EC2 instances in your own AWS account.** Maintain full control of your Amazon EC2 instances and Auto Scaling groups. This means that you can set up instance launch templates, maintain Auto Scaling group configurations, and integrate with other AWS services. As part of its Spot rebalancing activity, GameLift FleetIQ makes periodic updates to some Auto Scaling group properties. You can temporarily override these settings or suspend GameLift FleetIQ activity as needed.
• **Support for multiple game server executable formats.** GameLift FleetIQ supports all formats that currently run on Amazon EC2, including Windows, Linux, containers, and Kubernetes. See the Amazon EC2 FAQs for a list of supported operating systems and runtimes.

• **Multiple types of hosting resources.** With GameLift FleetIQ, you have access to more than 50 instance types for game server hosting (availability varies by Region). This means that you can pair your game server with the appropriate mix of CPU, memory, storage, and networking capacity to provide the best possible gaming experience for your players.

• **Worldwide reach.** GameLift FleetIQ is available in 15 Regions, including China. With this reach you can make your game servers accessible to players with minimal game lag, wherever they're located. See GameLift Service Endpoints for a complete list of Regions.

### How GameLift FleetIQ works

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

This topic provides a general overview of the GameLift FleetIQ standalone solution. To learn more about other GameLift solutions, see What Is Amazon GameLift? (p. 1).

GameLift FleetIQ is a game hosting layer that supplements the full set of computing resource management tools that you get with Amazon EC2 and Auto Scaling. Use GameLift FleetIQ to directly manage your Amazon EC2 resources, dynamically manage your Auto Scaling groups' capacity, and optionally integrate with other AWS services. GameLift FleetIQ provides an extra layer of logic that makes it possible to use low-cost Spot Instances for game hosting while also managing game hosting tasks such as tracking game server status and routing players to viable game servers.

With GameLift FleetIQ, you prepare as usual to launch Amazon EC2 instances for hosting: make an Amazon Machine Image (AMI) with your game server software and create an Amazon EC2 launch template. You also define configuration settings for an Auto Scaling group. When you're ready to set up your Auto Scaling group, you create a GameLift FleetIQ game server group. This action prompts GameLift FleetIQ to create both a game server group and a corresponding Auto Scaling group (using the AMI defined in the launch template) and link the two groups together.

When the Auto Scaling group is created, you have full access to your Amazon EC2 and Auto Scaling resources. You can change the configuration of your Auto Scaling groups, add optional Auto Scaling features such as multi-level scaling policies or load balancers, or integrate with other AWS services. You can connect directly to instances in the group. GameLift FleetIQ makes periodic updates to certain Auto Scaling group properties as part of its instance-balancing logic; you are free to change these values in the short term or you can temporarily suspend GameLift FleetIQ activity at any time. You also have the option to delete a game scaling group but retain the corresponding Auto Scaling group.

### GameLift FleetIQ logic

The following diagram illustrates the role of GameLift FleetIQ when it is working with Amazon EC2 for game hosting. Its primary goal is to locate the best possible game server to host a game session and give players an optimal gameplay experience. For game hosting, the best resources are defined as those that deliver the highest hosting viability for the lowest cost. GameLift FleetIQ approaches this goal in two key ways: by allowing only optimal instances types in the Auto Scaling group and by placing new game sessions effectively across the group's available resources.
Fill Auto Scaling group with optimal instance types

The Auto Scaling group’s job is to launch new instances and retire old instances, maintaining a collection of hosting resources and scaling it to meet your player demand. When you create a game server group, you create a list of desired instance types for your game. This list is passed to the Auto Scaling group, and every new instance that it launches uses an instance type from this list. GameLift FleetIQ affects how the Auto Scaling group selects optimal instance types in the following ways:

- When setting up the Auto Scaling group, GameLift FleetIQ configures the group to launch low-cost Spot Instances only. Spot Instances use spare computing capacity; they have lower costs due to fluctuating availability and potential interruptions. GameLift FleetIQ minimizes these limitations for game server hosting. You can optionally configure the Auto Scaling group to allow On-Demand Instances when there are no Spot Instances available.
- GameLift FleetIQ periodically evaluates each Spot Instance type on the game server group’s list to assess its viability for game hosting. This process uses a prediction algorithm that evaluates the instance type’s recent availability and interruption rate. If the chance of game server interruption is too high, GameLift FleetIQ removes this non-viable instance type from the Auto Scaling group’s list of desired instance types.
- GameLift FleetIQ identifies existing instances in the Auto Scaling group that are currently non-viable instance types. These instances are flagged as draining, which means they will be terminated and replaced with new instances. For instances that have game server protection turned on, termination is postponed until any active game sessions end normally.
- GameLift FleetIQ ensures that the Auto Scaling group instances are balanced across all viable instance types on the desired list. Groups with a diversity of instance types are more resilient and able to handle unexpected hosting viability issues.

As the Auto Scaling group launches and retires instances, it continually maintains a collection that is optimized for game hosting even as the availability of low-cost Spot Instance types fluctuates. Learn more about how this process works in Spot Rebalancing Process (p. 197).

Place game sessions effectively

GameLift FleetIQ tracks all active game servers in the game server group and uses this information to determine the best placement for new game sessions and players.

To enable GameLift FleetIQ to track game servers, your game server software must report its status. Your custom AMI controls how new game servers processes are started and stopped on each instance. When a new game server is started, it registers with GameLift FleetIQ, indicating that it is ready to host a game session. After registering, the game server periodically reports its health and whether it is currently hosting a game session. When the game server shuts down, it de-registers with GameLift FleetIQ.

To start a new game session, your game client (or matchmaker or other client service) sends a request for a game server to GameLift FleetIQ. GameLift FleetIQ locates an available game server, claims it for the new game session, and responds with the game server ID and connection info. Your game then prompts the game server to update its status and start a new game session for incoming players.

When selecting a game server to host a new game session, GameLift FleetIQ uses the following decision-making process to optimize placement with viable low-cost Spot Instances:

1. GameLift FleetIQ ignores instances that are flagged as draining, that is, not viable for game hosting. These instances are kept running only to support existing game sessions. They can't be used for new game sessions unless no other game servers are available.
2. GameLift FleetIQ identifies all available game servers that are running on viable instances.
3. Where possible, GameLift FleetIQ places new game sessions on instances that are already hosting other game sessions. By packing (but not overloading) some instances and keeping others idle, the
Auto Scaling group is able to quickly scale down idle instances when they're not needed, which lowers hosting costs.

You can turn on game session protection for a game server group to prevent the Auto Scaling group from terminating instances with actively running game sessions.

Key resources and components

Create the following resources in your AWS account before you set up your game hosting resources with GameLift FleetIQ. As a best practice, develop and test your game server deployment with these resources before using them through a game server group.

- **Amazon Machine Image (AMI).** An AMI is a template for a specific software configuration that you want to launch with your Amazon EC2 instances. For game hosting, your AMI includes an operating system, your game server binaries or container, and other runtime software that your game server requires. For more information about creating an AMI, refer to [Amazon Machine Images in the Amazon EC2 User Guide](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html) for Linux Instances. AMIs are Region-specific. You can copy an AMI from one Region to another, as described in [Copying AMIs in the Amazon EC2 User Guide](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html)

- **Amazon EC2 launch template.** A launch template provides instructions for launching and managing instances in an Auto Scaling group. It specifies an AMI, provides a list of suitable instance types, and sets network, security, and other properties. For more information about creating a launch template, see [Launching an Instance from a Launch Template](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/launch-template.html) in the [Amazon EC2 User Guide](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/launch-template.html). Launch templates are Region-specific.

- **AWS IAM role.** An IAM role defines a set of permissions that allow limited access to AWS resources. A trusted entity, such as another AWS service, can assume the role and inherit its permissions. When using GameLift FleetIQ, you must provide an IAM role with a managed policy that allows GameLift FleetIQ to create and access Auto Scaling groups and EC2 instance resources in your AWS account. IAM roles are not Region-specific.

GameLift FleetIQ manages the following resources directly and has direct authority over them.

- **GameLift game server group.** A game server group contains configuration settings that define how GameLift FleetIQ works with a corresponding Auto Scaling group to deliver low-cost game hosting. Game server groups are Region-specific. When you create a game server group in a Region, a new Auto Scaling group is automatically created in your AWS account in the same Region. The game server group is linked to the Auto Scaling group and has access (by assuming the IAM role) to manage and modify some of its settings. A game server group is a long-lived resource; developers should expect to create them infrequently. A game server group is also a functional grouping resource for game servers that are hosted on instances in the Auto Scaling group and registered with GameLift FleetIQ.

- **GameLift game server.** A game server resource represents a game execution that is running on an instance associated with a GameLift FleetIQ game server group. This resource is created when a game server registers with GameLift FleetIQ and identifies the game server group it belongs to. GameLift FleetIQ tracks the utilization status and claim status of each registered game server, which enables it to monitor game server availability. Game servers are Region-specific in that they are associated with a Region-specific game server group. When your game requests a new game server, it specifies the game server group and Region.

These resources are created through GameLift FleetIQ resources. They are created in your AWS account and you have full control of them.

- **Amazon EC2 Auto Scaling group.** An Auto Scaling group launches and manages a collection of EC2 instances, and automatically scales group capacity. With GameLift FleetIQ, there is a one-to-one relationship between the game server group and the Auto Scaling group. While you can update all settings for an Auto Scaling group, GameLift FleetIQ periodically overrides and updates certain
settings as part of its logic to balance Spot Instances for game hosting viability. For more information, see AutoScalingGroup in the Amazon EC2 Auto Scaling User Guide. Auto Scaling groups are Region-specific; they are created in the same Region as the game server group.

- **Amazon EC2 Instance.** An instance is a virtual server in the cloud. Instance types have specific hardware configurations that specify compute, memory, disk, and network resources. They are typically launched by an Auto Scaling group with an AMI. Instances can be Spot or On-Demand, depending on availability. With GameLift FleetIQ, instances run one or multiple game server processes, each of which can host multiple game sessions. Instances are Region-specific in that they are associated with a Region-specific Auto Scaling group.

## Game Architecture with GameLift FleetIQ

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

### Supplementing On-Premises Hosting

GameLift FleetIQ is designed to reuse your existing game backend, including any player geo-IP routing, matchmaking, or lobby services that you might already have in place. The following example illustrates how GameLift FleetIQ might fit into an existing on-premises deployment.

**Example**

In this example, game hosting is initially handled with four proprietary data centers to host players in North America and Europe. Depending on their approximate physical location, players are routed to one of two regional matchmakers. The matchmakers group players by skill and latency and then place them onto nearby game servers to minimize lag.

The game developer wants to replace their North America game servers with servers provided by GameLift FleetIQ. To start, they make minor updates to their game server to enable it for use with GameLift FleetIQ and then create an Amazon Machine Image (AMI). This image will be installed on every
EC2 instance that is deployed for the game. The image contains the game server, dependencies, and anything else needed to run game sessions for players.

With the AMI ready, the developer creates two GameLift FleetIQ game server groups, one for each AWS North America Region (`us-east-1` and `us-west-2`). The developer passes in the AMI, a list of desired instance types, and other configuration settings for the group. The list of desired instance types tells GameLift FleetIQ which types to use when searching for Spot Instances that are viable for game hosting.

Finally, the developer integrates the AWS SDK with GameLift FleetIQ into their North American matchmaker, which calls GameLift FleetIQ when a new group of players needs server capacity for a game session. GameLift FleetIQ locates a Spot instance with an available game server, reserves it for the players, and provides server connection information. Players connect to the server, play the game, and disconnect. To start a new game, players re-enter matchmaking, which prompts GameLift FleetIQ to find another available game server. Each new game request triggers GameLift FleetIQ to search for and select game servers with a low chance of interruptions. As a result, GameLift FleetIQ is constantly redirecting players away from game servers that are not viable for game hosting, even as Spot Instance availability fluctuates over time.

**GameLift FleetIQ Lifecycles and States**

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

**Spot Rebalancing Process**

GameLift FleetIQ periodically rebalances the instances in an Auto Scaling group. This process has two key goals:

- To constantly refresh the group by only using Spot Instance types that currently have high viability for game hosting.
- To use multiple viable instance types (where possible) in order to reduce the impact of unexpected game server interruptions.

GameLift FleetIQ rebalances by evaluating the group's instance types and removing instances that are more likely to result in game server interruptions. To avoid terminating instances with active gameplay during rebalancing, best practice is to turn on game server protection for a game server group that is in production.
Example

The following example illustrates how instances in an Auto Scaling group are affected by Spot rebalancing.

- **Step 1.** Through a game server group, the linked Auto Scaling group is set up to launch instances of types m4.large, c4.large, and c5.large with game server protection enabled. The Auto Scaling group has launched a balanced collection consisting of two Spot Instances of each type. Four instances have at least one game server in UTILIZED status (shown with a dashed border), while two instances are not currently supporting gameplay.

- **Step 2.** GameLift FleetIQ evaluates the current game hosting viability of all three instance types. The evaluation determines that the c4.large instance type has an unacceptable potential for game server interruption. GameLift FleetIQ immediately updates the Auto Scaling group configuration to temporarily remove c4.large from the list of instance types, preventing additional instances of that type from being launched.

- **Step 3.** GameLift FleetIQ identifies existing instances of type c4.large and takes actions to remove them from the group. As a first step, all game servers that are running on c4.large instances are flagged as *draining*. Game servers on draining instances can be claimed only as a last resort if no other game servers are available. In addition, an Auto Scaling group with draining instances is triggered to launch new instances to replace them.

- **Step 4.** As new viable instances come online, the Auto Scaling group terminates draining instances. This replacement ensures that the group's desired capacity is maintained. The first instance to be terminated is the c4.large instance with no utilized game servers and game server protection turned off. It is replaced with a new c5.large instance.

- **Step 5.** Draining instances with game server protection continue to run while their game servers are supporting gameplay. When gameplay ends, the remaining c4.large instance is terminated when a new m4.large instance has launched to take its place.

As a result of this process, the Auto Scaling group maintains its desired capacity while the group rebalances from using three instance types to two. GameLift FleetIQ continues to evaluate the original list of instance types for game hosting viability. When c4.large is again considered a viable instance type, the Auto Scaling group is updated to include all three instance types. The group naturally rebalances over time.
Life of a Game Server Group

Game server groups go through the following lifecycle, including provisioning and status updates. A game server group is expected to be a long-lived resource.

- You create a game server group by calling the GameLift API `CreateGameServerGroup()` and passing in an EC2 launch template and configuration settings. In response to the call, a new game server group is created and placed in status NEW.

- GameLift FleetIQ activates an asynchronous activation workflow, transitioning the game server group status to ACTIVATING. The workflow initiates the creation of underlying resources, including an Amazon EC2 Auto Scaling group and an EC2 instance with the provided AMI.
  - If provisioning fails for any reason, the game server group is placed into status ERROR. To get additional error information to help debug the failure cause, call `DescribeGameServerGroup()` on a game server group in an error state.

- If provisioning succeeds, the game server group is transitioned to status ACTIVE. At this point, instances are launched with game servers that register with GameLift FleetIQ. The group's instance types are periodically evaluated for game hosting viability and rebalanced as needed. GameLift FleetIQ also tracks the status of active game servers in the group and responds to requests for game servers.

- You remove a game server group by calling `DeleteGameServerGroup()` with the group identifier. This action puts the game server group into status DELETE_SCHEDULED. Only game server groups in ACTIVE or ERROR state can be scheduled for deletion.

- GameLift FleetIQ activates an asynchronous deactivation workflow in response to the DELETE_SCHEDULED status, transitioning the game server group status to DELETING. You have the option of deleting just the game server group or delete both the game server group and the linked Auto Scaling group.
  - If deactivation fails for any reason, the game server group is placed into status ERROR. To get additional error information to help debug the failure cause, call `DescribeGameServerGroup()` on a game server group in an error state.

  - If deactivation succeeds, the game server group is transitioned to status DELETED.

Life of a Game Server

With GameLift FleetIQ, game servers go through the following lifecycle, including provisioning and status updates. A game server is expected to be a short-lived resource. We recommend that game servers be de-registered after the end of a game session rather than being re-used. This approach continually refreshes the game server group and ensures that available game servers are always running on the lowest-cost resources that are viable for game hosting.
A game server resource is created when the game server process, running on an instance in a GameLift FleetIQ-linked Auto Scaling group, calls the GameLift API `RegisterGameServer()` to notify GameLift FleetIQ that it is ready to host players and gameplay. A game server has two statuses to track its current availability:

- **Utilization status** tracks whether the game server is currently supporting gameplay. This status is initially set to AVAILABLE, indicating that it is ready to accept new gameplay. Once the game server is occupied with gameplay, this status is set to UTILIZED.

- **Claim status** tracks whether the game server is claimed for imminent gameplay. A game server in CLAIMED status indicates that it has been temporarily reserved by a game client (or a game service such as a matchmaker). This status prevents GameLift FleetIQ from providing the same game server to multiple requesters. A game server with a blank claim status is available to be claimed.

The following diagram illustrates how a game server’s utilization status and claim status change over its life span.

**Step 1.** A game server group has six registered game servers. Four have a utilization status AVAILABLE (A, B, C, and D), and two are currently UTILIZED (E and F).

**Step 2.** A game client or matchmaking system calls the GameLift API `ClaimGameServer()` to request a new game server. This request prompts GameLift FleetIQ to search for an available game server (D) and set its claim status to CLAIMED for 60 seconds. GameLift FleetIQ responds to its request with connection information for the game server (IP address and port), as well as other optional game-specific data. Since gameplay has not yet begun on the game server, its utilization status remains AVAILABLE, but it cannot be claimed with another request.
• **Step 3a.** Using the connection information provided, game clients can connect to the game server and initiate gameplay. The game server (D) must be triggered within 60 seconds to change its utilization status to UTILIZED by calling the GameLift API `UpdateGameServer()`.

• **Step 3b.** If the game server's utilization status is not updated within 60 seconds, the claim timer expires and the claim status is reset to blank. The game server (D) is returned to the pool of available and unclaimed game servers.

• A game server resource is removed after gameplay on the game server is complete and players have disconnected. Before shutting down, the game server process calls the GameLift API `DeregisterGameServer()` to notify GameLift FleetIQ of its departure from the game server group's pool of game servers.

---

### GameLift FleetIQ Setting Up

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

This section describes the key procedures for setting up your game for the Amazon GameLift FleetIQ service. It includes how to get the necessary tools to use GameLift FleetIQ with Amazon EC2 and Auto Scaling to host your multiplayer games.

**Topics**

- GameLift FleetIQ SDKs and Platforms (p. 201)
- Setting up your AWS account for GameLift FleetIQ (p. 202)

### GameLift FleetIQ SDKs and Platforms

Create and deploy 64-bit, multiplayer game servers, clients, and game services for use with Amazon EC2 resources and GameLift FleetIQ.

**SDK support**

The AWS SDK with the GameLift API contains functionality that allows game servers and clients to communicate with the GameLift FleetIQ service and enables GameLift to optimize game session and player placement. Download the AWS SDK or view the Amazon GameLift API reference documentation.

The AWS SDK with Amazon GameLift is available in the following languages. See documentation for each language for details on support for development environments.

- [C++](https://docs.aws.amazon.com/sdkforcpp/index.html) (Amazon GameLift)
- [Java](https://docs.aws.amazon.com/sdkfortarget-language/latest API documentation) (Amazon GameLift)
- [.NET](https://docs.aws.amazon.com/sdkfortarget-language/latest API documentation) (Amazon GameLift)
- [Go](https://docs.aws.amazon.com/sdkfortarget-language/latest API documentation) (Amazon GameLift)
- [Python](https://docs.aws.amazon.com/sdkfortarget-language/latest API documentation) (Amazon GameLift)
- [Ruby](https://docs.aws.amazon.com/sdkfortarget-language/latest API documentation) (Amazon GameLift)
- [PHP](https://docs.aws.amazon.com/sdkfortarget-language/latest API documentation) (Amazon GameLift)
- [JavaScript/Node.js](https://docs.aws.amazon.com/sdkfortarget-language/latest API documentation) (Amazon GameLift)

**Operating systems and development environments**

Version

201
GameLift FleetIQ supports any operating system, engine, and programming language that runs on EC2. Amazon EC2 currently supports a variety of operating systems including: Amazon Linux, Ubuntu, Windows Server, Red Hat Enterprise Linux, SUSE Linux Enterprise Server, Fedora, Debian, CentOS, Gentoo Linux, Oracle Linux, and FreeBSD.

You can host game servers and integrate game clients and client services developed with Unreal, Unity, and custom game engines as well as game servers written in C++, C#, and other languages supported by the AWS SDK.

GameLift FleetIQ supports containers with Kubernetes, Amazon Elastic Container Service (Amazon ECS), and Amazon Elastic Kubernetes Service (EKS).

Setting up your AWS account for GameLift FleetIQ

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

When using GameLift FleetIQ with Amazon EC2, Auto Scaling and other AWS services, you use AWS Identity and Access Management to manage permissions across services. To set up your AWS account for GameLift FleetIQ, do the following:

- Create an AWS account for use with GameLift FleetIQ.
- Set up IAM permissions for users and user groups who work with the AWS services involved with your game.
- Create a managed IAM role that gives GameLift limited permissions to interact with your Amazon EC2 instances and Auto Scaling groups.

Get an account for use with Amazon GameLift

Open Amazon Web Services and choose Sign In to the Console. Follow the prompts to either create a new account or sign into an existing one.

Update IAM user and user group policies

Users who work with GameLift FleetIQ game server groups and the related Amazon EC2 and Auto Scaling services must have access to relevant actions in these services. To provide access, create users or user groups in AWS Identity and Access Management (IAM).

Open the IAM console to define a set of users or user groups for your game. For detailed instructions on using the console (or the AWS CLI or other tools) to set up your user groups, see Creating IAM Users. To extend permissions to a user or user group, attach an IAM policy, which specifies the set of AWS services and actions that a user can access. Create an inline custom policy with the following syntax, and attach it to each user or user group.

GameLift FleetIQ policy

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "iam:PassRole"
            ],
```
Create an IAM service role

As part of its Spot rebalancing activities, GameLift FleetIQ must access and modify the Auto Scaling groups, instances, and lifecycle hooks that are linked with your GameLift FleetIQ game server groups. To accomplish this, you create an IAM role that grants GameLift limited permissions for these actions. For more details, see Creating a Role for an AWS Service in the IAM User Guide.

Console

These steps describe how to create the necessary role using the AWS Management Console.

1. Open the IAM console and choose Roles: Create role.
2. For Type of trusted entity, choose AWS service.
3. For Use case, choose GameLift from the list of services. This action automatically selects the appropriate use case. To proceed, click Next: Permissions.
4. The policy GameLiftGameServerGroupPolicy should be automatically listed. If this policy is not shown, use the search feature to add it to the role. You can view the specific policy syntax by expanding the policy information. To proceed through the remaining workflow, click Next.
5. On the **Review** page, enter a role name and optionally update the description. Verify that the new role specifies the trusted entity **AWS service: gamelift.amazonaws.com**, and includes the policy **GameLiftGameServerGroupPolicy**. When ready, click **Create role**.

**AWS CLI**

These steps describe how to create the necessary role using the AWS CLI.

1. Create two JSON files, one for the trust policy, and one for the permissions policy. Copy the syntax shown below.

2. Create a new IAM role and associate it with the trust policy JSON.

   ```bash
   aws iam create-role --role-name Role-for-GameLift --assume-role-policy-document file://C:\policies\trustpolicyforGameLift.json
   ```

   If you’re working on Linux, reference the JSON file like this: `file://policies/trustpolicyforfiesta.json`

3. Update the IAM role with the permissions policy JSON.

   ```bash
   aws iam put-role-policy --role-name Test-Role-for-Fiesta --policy-name Permissions-Policy-For-GameLift --policy-document file://C:\policies\permissionspolicyforGameLift.json
   ```

   If you’re working on Linux, reference the JSON file like this: `file://policies/permissionspolicyforfiesta.json`

4. To test if you have configured the IAM role correctly, call the **CreateGameServerGroup** API, passing in your IAM role. If the **GameServerGroup** enters an **ACTIVE** state, GameLift FleetIQ can modify AWS resources in your account as expected.

**Trust policy syntax**

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "gamelift.amazonaws.com",
          "autoscaling.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

**Permissions policy syntax**

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "ec2:TerminateInstances",
      "Resource": "*"
    }
  ]
}
```
Getting Started with GameLift FleetIQ

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

To get started using GameLift FleetIQ, you should have a working game server that runs either in an on-premises or EC2 environment. Your game server can be a single process that manages one or multiple game sessions, spawns child processes, or runs inside of a container.

1. Get an AWS account.

If you already have one or more AWS accounts, choose the AWS account that you want to use for GameLift FleetIQ. This account and user/user group must have permissions to manage your Amazon EC2, Auto Scaling, and other AWS resources used with your game. For more information, see Update
IAM user and user group policies (p. 202). Later in this getting started guide you create an IAM role to manage permissions across services.

2. Get the AWS SDK and AWS CLI with GameLift FleetIQ functionality.
   - Download the latest version of the AWS SDK.
   - Get and install the AWS Command Line Interface tool.
   - View the Amazon GameLift API reference documentation.

3. Prepare your game server for use with GameLift FleetIQ.

   Integrate the AWS SDK into your game server and implement the following three APIs:
   - RegisterGameServer(): called once when the game server process starts.
   - UpdateGameServer(): called periodically to report health and the presence of players in your game server.
   - DeregisterGameServer(): called once when the game server is terminated.

   These APIs keep GameLift FleetIQ updated with the state of your game servers (whether they are available to host players or are occupied). With this information, GameLift FleetIQ can provide your matchmaking system with a list of viable, unoccupied game servers and also avoid terminating instances with players on them during rebalancing. See the section called “Integrate a Game Server” (p. 207) for additional guidance and examples.

4. Create an EC2 Amazon Machine Image (AMI) with your game server.

   Create an AMI with your game server software and with any other runtime assets or configuration settings. For help, see Amazon Machine Images (AMI) in the Amazon EC2 User Guide.

5. Create an EC2 launch template.

   Build an EC2 launch template that uses your custom AMI and defines network and security settings for your hosting resources. You don’t need to include instance types in your launch template, as this is done later. For help, see Creating a Launch Template in the Amazon EC2 User Guide. We highly recommend creating an Auto Scaling group directly to test your launch template and AMI before using it with GameLift FleetIQ to simplify debugging tasks.

6. Create an IAM role.

   Give the GameLift service permission to create and update Auto Scaling groups in your account. GameLift FleetIQ uses this IAM role to update the instances in your Auto Scaling group as part of instance type rebalancing. See Setting up your AWS account for GameLift FleetIQ (p. 202) for additional instructions and IAM policy syntax.

7. Set up GameLift FleetIQ hosting resources.

   In each Region where you want to deploy game servers, create a game server group by calling CreateGameServerGroup(). Pass in the launch template (containing your custom AMI and network and security settings), IAM role, and a list of instance types that your game can run on. This action sets up an Auto Scaling group in your AWS account that GameLift FleetIQ can modify. For additional guidance and examples, see Create and Update Game Server Groups (p. 209).

8. Integrate GameLift FleetIQ into your game client.

   Add the AWS SDK to your game client, matchmaker, or other backend component that allocates game server capacity. Depending on your game type, your matchmaker might call ListGameServers() or ClaimGameServer() to obtain server capacity and reserve an available game server. For additional guidance and examples, see Integrate GameLift FleetIQ into a Game Client (p. 208).

9. Scale up your Auto Scaling group.

   As instances are provisioned in your Auto Scaling group, they launch your game servers. Each game server then registers with GameLift FleetIQ as available capacity, to be listed or claimed later by your matchmaker.
10 Test your game.

Invoke your matchmaker and call ListGameServers or ClaimGameServer to request server capacity. Pass the resulting IP and port back to game clients so they can connect to the game server.

Integrate GameLift FleetIQ into a Game Server

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

When a game server process is launched and ready to host live gameplay, it should register with GameLift FleetIQ by calling RegisterGameServer(). Registering allows GameLift FleetIQ to respond to matchmaking system requests for server capacity. Responses include available game server capacity on Spot Instances that are viable for game hosting. When registering, the game server can provide GameLift FleetIQ with relevant game server data and connection information, including the port and IP address that it uses for inbound client connections.

```
aws gamelift register-game-server \
  --game-server-id UniqueId-1234 \
  --game-server-group-name MyLiveGroup \
  --instance-id i-1234567890 \
  --connection-info "1.2.3.4:123" \
  --game-server-data "{"key": "value"}"
```

After registering, your game server should regularly report its health and whether it is occupied by players to keep the GameLift FleetIQ state in sync. You report health and utilization status by calling UpdateGameServer(). In the example below, the game server is reporting that it is healthy and is not yet occupied by players or gameplay.

```
aws gamelift update-game-server \
  --game-server-group-name MyLiveGroup \
  --game-server-id UniqueId-1234 \
  --health-check HEALTHY \
  --utilization-status AVAILABLE
```

When your matchmaking system requests server capacity, it calls ClaimGameServer() to prompt GameLift FleetIQ to locate an available game server on a Spot Instance that is viable for game hosting. The response identifies a GameServer resource with its claim status set to CLAIMED for 60 seconds. It might also contain connection information and game data, which clients can use to connect to the game server.

Your game must have a mechanism, triggered by your game client, client service, or game server, that updates the game server’s status after players connect and gameplay starts. This is done by calling UpdateGameServer() with utilization status set to UTILIZED to indicate that the game is now supporting gameplay. This mechanism ensures that GameLift FleetIQ does not provide this game server in response to subsequent requests for server capacity.

```
aws update-game-server \
  --game-server-group-name MyLiveGroup \
  --game-server-id UniqueId-1234 \
  --health-check HEALTHY \
  --utilization-status UTILIZED
```

When a game concludes, the game server should deregister from GameLift FleetIQ using DeregisterGameServer().
Integrate GameLift FleetIQ into a Game Client

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

Create a method that allows your game client or matchmaker to request a game server resource for players. You have a couple of options for how to do this:

- Request all available game servers and select one to use (often referred to as "list and pick").
- Have GameLift FleetIQ choose an available game server.

To use the list and pick method, have your game client or matchmaker request a list of available game servers by calling `ListGameServers()`. You might want to use game server data to provide additional information that players or your matchmaker can use when selecting a game server. You can request paginated results and use custom sort keys to control how results are returned. The following request returns all active and available game servers in the specified game server group.

```
aws gamelist list-game-servers \
  --game-server-group-name MyLiveGroup
```

Players or your matchmaker can select a suitable game server and claim it by calling `ClaimGameServer()` with the specific game server ID. In this scenario, GameLift FleetIQ does not exercise any of its instance type optimization logic.

```
aws GameLift FleetIQ claim-game-server \
  --game-server-group-name MyLiveGroup \
  --client-token 12345 \
  --game-server-id UniqueId-1234
```

To have GameLift FleetIQ choose an available game server, call `ClaimGameServer()` without specifying a game server ID. In this scenario, GameLift FleetIQ does exercise its logic to find a game server on an instance that is viable for game hosting and optimized for automatic scaling.

```
aws GameLift FleetIQ claim-game-server \
  --game-server-group-name MyLiveGroup \
  --client-token 12345
```

In response to a claim request, GameLift FleetIQ identifies the `GameServer` resource, connection information, and game data, which clients can use to connect to the game server. The game server's claim status is set to CLAIMED for 60 seconds. Your game must have a mechanism, triggered by your game client, client service, or game server, that updates the game server's status after players connect and gameplay starts. This is done by calling `UpdateGameServer()` with utilization status set to UTILIZED to indicate that the game is now supporting gameplay. This mechanism ensures that GameLift FleetIQ does not provide this game server in response to subsequent requests for game server capacity.

```
aws update-game-server \
  --game-server-group-name MyLiveGroup \
  --game-server-id UniqueId-1234
```
Create and Update Game Server Groups

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

To create a game server group, call `CreateGameServerGroup()`. In the following example, a `GameServerGroup` is created that specifies `c4.large` and `c5.large` instance types for the Auto Scaling group, and limits the group to Spot Instances only. After a short provisioning period, an `AutoScalingGroup` resource is created, and the `GameServerGroup` enters an ACTIVE state.

```
aws gamelift create-game-server-group \
  --game-server-group-name MyLiveGroup \
  --role-arn arn:aws:iam:: 123456789012 :role/GameLiftGSGRole \
  --min-size 1 \
  --max-size 10 \
  --game-server-protection-policy FULL_PROTECTION \
  --balancing-strategy SPOT_ONLY \
  --launch-template LaunchTemplateId=lt-012ab345cde6789ff \
  --instance-definitions '[[{"InstanceType": "c4.large"}],{"InstanceType": "c5.large"}]'] \
  --auto-scaling-policy '{"TargetTrackingConfiguration":{"TargetValue": 66}}'
```

Instance type properties for the Auto Scaling group are periodically overwritten by GameLift FleetIQ and should not be directly updated on the Auto Scaling group. To update properties that GameLift FleetIQ overwrites, call `UpdateGameServerGroup()`. After changes are made to the game server group, GameLift FleetIQ propagates changes into the Auto Scaling group. Other properties, such as `MinSize`, `MaxSize`, and `LaunchTemplate`, can be modified directly on the Auto Scaling group.

In the example below, the instance types are updated to switch over to `c4.xlarge` and `c5.xlarge` instances types.

```
aws fiesta update-game-server-group \
  --game-server-group-name MyLiveGroup \
  --instance-definitions '[[{"InstanceType": "c4.xlarge"}],{"InstanceType": "c5.xlarge"}]'
```

GameLift FleetIQ Best Practices

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

GameLift FleetIQ is a low-level logic layer that helps you manage low-cost Amazon EC2 resources for game hosting. GameLift FleetIQ optimizes the game server group to contain only Spot Instances that are viable for game hosting, minimizing the chance that game sessions might be interrupted. It also provides basic game hosting functionality to track available game servers and route gameplay to low-cost, high-viability game servers.

GameLift FleetIQ as a standalone feature does not provide advanced features that are offered with the full GameLift managed solution, which also uses FleetIQ to minimize hosting costs. If you need features
such as matchmaking, latency-based player routing, game session and player session management, and versioning, take a look at the GameLift managed solution.

Here are some best practices that can help you get the most benefit from GameLift FleetIQ.

- **Use GameLift FleetIQ for session-based games.** GameLift FleetIQ works best when it is constantly directing players onto instances that are least likely to have game session interruptions. Maintaining long-lived sessions interferes with the GameLift FleetIQ rebalancing process, which increases the likelihood that games sessions might be interrupted. The ideal workflow is for players to go from matchmaking (or server selection) into gameplay. When the game ends, players go back to matchmaking and are routed to another game server on a new instance. We recommend using GameLift FleetIQ for games with sessions under two hours.

- **Provide many instance types to choose from.** When you set up a game server group, you provide a list of instance types to be used. The more instance types that you include, the greater flexibility GameLift FleetIQ has to use Spot Instances with high viability for game hosting. For example, you might list multiple sizes within the same instance family (c5.large, c5.xlarge, c5.2xlarge, c5.4xlarge). With larger instances, you can run more game servers on each instance, potentially lowering costs. With smaller instances, autoscaling can react faster to changes in player demand. Keep in mind that the list of desired instance types is not prioritized—an Auto Scaling group will use a balance of viable instance types to maintain the group's resiliency.

- **Test your game on all instance types.** Ensure that your game server runs properly on every instance type that you configure for your game server group.

- **Use instance capacity weighting.** If you configure your game server group to use a range of instance sizes (such as c5.2xlarge, c5.4xlarge, c5.12xlarge), include capacity weighting information for each instance type. For more information, see Instance Weighting for Amazon EC2 Auto Scaling in the Amazon EC2 Auto Scaling User Guide.

- **Place your game sessions using GameLift FleetIQ.** When placing groups of players with game servers, use the GameLift API `ClaimGameServer()`. GameLift FleetIQ avoids placing players on instances with a higher chance of game session interruptions.

- **Report game server status to GameLift FleetIQ.** Periodically report server health and utilization status with the GameLift API `UpdateGameServer()`. Maintaining accurate game server status helps GameLift FleetIQ place gameplay more efficiently. It also helps avoid terminating instances with active gameplay during Spot rebalancing activity.

- **Set up an autoscaling policy.** You can create a target-tracking scaling policy that maintains your hosting capacity based on player utilization and anticipated demand. The GameLift FleetIQ metric `PercentUtilizedGameServers` is a measure of how much of your hosting capacity is currently in use. Most games want to maintain a buffer of unused game servers so that new players can get into a game quickly. You can create a scaling policy that maintains a certain buffer size, adding or removing instances as player demand fluctuates. For more information, see Target Tracking Scaling Policies in the Amazon EC2 Auto Scaling User Guide.

- **Use different AWS accounts for development and production environments.** Separating your development and production configurations across accounts can reduce the risk of misconfiguration impacting live players.

- **Enable game session protection for game server groups in production.** To protect your players, turn on game session protection and prevent active game sessions from being terminated early due to scaling or rebalancing activity.

- **Test your game on EC2 before integrating it with GameLift FleetIQ.** We recommend getting your game up and running on EC2 and fine-tune your configuration first. You can then create a game server group using the same launch template and AMI.

If you're using Kubernetes, we recommend first getting standard EC2 instances added to your Kubernetes cluster, then create a game server group using the launch template you create for worker nodes in your Kubernetes cluster. If you're using EKS, create your EKS cluster and game server group separately. For the game server group, use the EKS-optimized AMI with the appropriate user data and...
the launch template configuration used for your EKS integration. See more details about EKS worker nodes and the EKS optimized AMI in the Amazon EKS-Optimized Linux AMI guide.

Also review these AWS best practices:

- Best Practices for Amazon EC2
- Best Practices for Amazon EC2 Auto Scaling

## Monitor GameLift FleetIQ with Amazon CloudWatch

Amazon GameLift FleetIQ with game server groups is in preview release and is subject to change. FleetIQ as a feature of managed GameLift is unchanged.

GameLift FleetIQ emits five Amazon CloudWatch metrics to your AWS account. These metrics can help you to scale your instance capacity, build operations dashboards, and trigger alarming.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AvailableGameServers</td>
<td>Game servers that are available to run a game execution and are not currently occupied with gameplay. This number includes game servers that have been claimed but are still in AVAILABLE status.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td></td>
<td>Dimensions: GameServerGroup</td>
</tr>
<tr>
<td>UtilizedGameServers</td>
<td>Game servers that are currently occupied with gameplay. This number includes game servers that are in UTILIZED status.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td></td>
<td>Dimensions: GameServerGroup</td>
</tr>
<tr>
<td>DrainingAvailableGameServers</td>
<td>Game servers on instances scheduled for termination that are currently not supporting gameplay. These game servers are the lowest priority to be claimed in response to a new claim request.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td></td>
<td>Dimensions: GameServerGroup</td>
</tr>
<tr>
<td>DrainingUtilizedGameServers</td>
<td>Game servers on instances scheduled for termination that are currently supporting gameplay.</td>
</tr>
</tbody>
</table>

Version 211
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td></td>
<td>Dimensions: GameServerGroup</td>
</tr>
<tr>
<td>PercentUtilizedGameServers</td>
<td>Portion of game servers that are currently supporting game executions. This metric indicates the amount of game server capacity that is currently in use. It is useful for driving an Auto Scaling policy that can dynamically add and remove instances to match with player demand.</td>
</tr>
<tr>
<td></td>
<td>Units: Percent</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Average, Minimum, Maximum</td>
</tr>
<tr>
<td></td>
<td>Dimensions: GameServerGroup</td>
</tr>
<tr>
<td>GameServerInterruptions</td>
<td>Game servers on Spot Instances that were interrupted due to limited Spot availability.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td></td>
<td>Dimensions: GameServerGroup, InstanceType</td>
</tr>
<tr>
<td>InstanceInterruptions</td>
<td>Spot Instances that were interrupted due to limited availability.</td>
</tr>
<tr>
<td></td>
<td>Units: Count</td>
</tr>
<tr>
<td></td>
<td>Relevant CloudWatch statistics: Sum</td>
</tr>
<tr>
<td></td>
<td>Dimensions: GameServerGroup, InstanceType</td>
</tr>
</tbody>
</table>
Amazon GameLift Reference Guides

This section contains reference documentation for using Amazon GameLift.

Topics

- Amazon GameLift Service API Reference (AWS SDK) (p. 213)
- Amazon GameLift Realtime Servers Reference (p. 217)
- Amazon GameLift Server SDK Reference (p. 232)
- Amazon GameLift FlexMatch Reference (p. 269)

Amazon GameLift Service API Reference (AWS SDK)

The Amazon GameLift Service API is packaged into the AWS SDK. Download the AWS SDK or view the Amazon GameLift API reference documentation.

APIs for Managed GameLift

These APIs are part of the full managed GameLift solution for custom game servers and Realtime Servers. They are organized here into two categories:

- APIs to Manage Games and Players (p. 213)
- Actions to Manage Game Hosting Resources (p. 214)

APIs to Manage Games and Players

Call these API actions from your game client service to start new game sessions, request matchmaking, reserve player slots in active games, and work with game and player session data.

- **Start new game sessions for one or more players.** Use a game session placement to start a new game and place it with the best hosting resources available. Alternatively, create a new game session on one specific fleet.
  - StartGameSessionPlacement – Request a new game session placement and add one or more players to it.
  - DescribeGameSessionPlacement – Get details on a placement request, including status.
  - StopGameSessionPlacement – Cancel a placement request.
  - CreateGameSession – Start a new game session on a specific fleet. Available in GameLift Local.
- **Get players into game sessions with FlexMatch matchmaking.** Group players into a match and start a new game session for them. Find new players for an existing game using match backfill.
  - StartMatchmaking – Request matchmaking for one players or a group who want to play together.
  - DescribeMatchmaking – Get details on a matchmaking request, including status.
  - AcceptMatch – For a match that requires player acceptance, register 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.
• Get players into existing games. Find existing games with available player slots and reserve them for new players.
  • SearchGameSessions – Retrieve all available game sessions or search for game sessions that match a set of criteria.
  • CreatePlayerSession – Reserve an open slot for a player to join a game session. Available in GameLift Local.
  • CreatePlayerSessions – Reserve open slots for multiple players to join a game session. Available in GameLift Local.

• Work with game session and player session data. Retrieve current data on existing game sessions and player sessions, and update as needed.
  • DescribeGameSessions – Retrieve metadata for one or more game sessions, including length of time active and current player count. Available in GameLift Local.
  • DescribeGameSessionDetails – Retrieve metadata and the game session protection setting for one or more game sessions.
  • GetGameSessionLogUrl – Get the location of saved logs for a game session.
  • DescribePlayerSessions – Get details on player activity, including status, playing time, and player data. Available in GameLift Local.
  • UpdateGameSession – Change game session settings, such as maximum player count and join policy.

**Actions to Manage Game Hosting Resources**

Use these API actions to configure hosting resources for your game, scale capacity to meet player demand, access performance and utilization metrics, and more. Most resource management functionality is available in the GameLift Console, but you can also make direct calls to the service using the AWS Command Line Interface (AWS CLI) tool or the AWS SDK.

• Manage game builds. Work with builds of your custom game server, which are uploaded to the GameLift service and deployed on fleets.
  • CreateBuild – Create a new build using files stored in an Amazon S3 bucket. To create a build and upload files from a local path, use the AWS CLI-only command upload-build.
  • ListBuilds – Get a list of all builds uploaded to a GameLift region.
  • DescribeBuild – Retrieve information associated with a build.
  • UpdateBuild – Change build metadata, including build name and version.
  • DeleteBuild – Remove a build from GameLift.

• Manage Realtime scripts. Work with configuration scripts for use with Realtime Servers. Unlike custom game builds, scripts can be updated after they are uploaded to the GameLift service.
  • CreateScript – Create a new server script to run on Realtime Servers.
  • ListScripts – Get a list of all Realtime scripts uploaded to an GameLift region.
  • DescribeScript – Retrieve information associated with a Realtime script.
  • UpdateScript – Change script metadata and upload revised script content.
  • DeleteScript – Remove a Realtime script from GameLift.

• Manage fleets for game hosting. Configure a fleet of hosting resources and deploy your game server or Realtime script to the fleet.
  • CreateFleet – Configure and activate a new fleet to run your custom game server build or Realtime script.
  • ListFleets – Get a list of all fleets in a GameLift region.
  • DeleteFleet – Terminate a fleet that is no longer running game servers or hosting players.
  • View / update fleet configurations.
  • DescribeFleetAttributes / UpdateFleetAttributes – View or change a fleet’s metadata and settings for game session protection and resource creation limits.
• **DescribeFleetPortSettings / UpdateFleetPortSettings** – View or change the inbound permissions (IP address and port setting ranges) allowed for a fleet.

• **DescribeRuntimeConfiguration / UpdateRuntimeConfiguration** – View or change what server processes (and how many) to run on each instance in a fleet.

• **Scale fleet capacity.** Set up fleet automatic scaling or set fleet capacity manually.
  
  • **DescribeEC2InstanceLimits** – Retrieve maximum number of instances allowed for the current AWS account and the current usage level.
  
  • **DescribeFleetCapacity** – Retrieve the fleet's current capacity settings.
  
  • **UpdateFleetCapacity** – Manually adjust fleet capacity settings.

• Manage auto-scaling.
  
  • **PutScalingPolicy** – Turn on target-based auto-scaling or create a custom auto-scaling policy, or update an existing policy.
  
  • **DescribeScalingPolicies** – Retrieve an existing auto-scaling policy.
  
  • **DeleteScalingPolicy** – Delete an auto-scaling policy and stop it from affecting a fleet's capacity.
  
  • **StartFleetActions** – Restart a fleet's auto-scaling policies.
  
  • **StopFleetActions** – Suspend a fleet's auto-scaling policies.

• **Manage game session queues.** Set up multi-fleet, multi-region Queues to place game sessions with the best available hosting resources. Queues are required with FlexMatch matchmaking.
  
  • **CreateGameSessionQueue** – Create a queue for use when processing requests for game session placements.
  
  • **DescribeGameSessionQueues** – Retrieve game session queues defined in a GameLift region.
  
  • **UpdateGameSessionQueue** – Change the configuration of a game session queue.
  
  • **DeleteGameSessionQueue** – Remove a game session queue from the region.

• **Manage FlexMatch resources.** Configure matchmakers for your game and specify match rules to create teams of players to meet your custom specifications.
  
  • **CreateMatchmakingConfiguration** – Create a matchmaking configuration with instructions for building a player group and placing in a new game session.
  
  • **DescribeMatchmakingConfigurations** – Retrieve matchmaking configurations defined for a GameLift region.
  
  • **UpdateMatchmakingConfiguration** – Change settings for matchmaking configuration.
  
  • **DeleteMatchmakingConfiguration** – Remove a matchmaking configuration from the region.
  
  • **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.

• **Monitor fleet activity.** Get up-to-the-minute information on server process and game session activity on a fleet.
  
  • **DescribeFleetUtilization** – Retrieve statistics on the number of server processes, game sessions, and players that are currently active on a fleet.
  
  • **DescribeFleetEvents** – View logged events for a fleet during a specified time span.
  
  • **DescribeGameSessions** – Retrieve game session metadata, including a game's running time and current player count.

• **Remotely access an instance.** Monitor or troubleshoot activity on a specified fleet instance.
  
  • **DescribeInstances** – Get information on each instance in a fleet, including instance ID, IP address, and status.
  
  • **GetInstanceAccess** – Request access credentials needed to remotely connect to a specified instance in a fleet.

• **Manage fleet aliases.** Use aliases to represent your fleets or specify an alternative destination.
• **CreateAlias** – Define a new alias and optionally assign it to a fleet.
• **ListAliases** – Get all fleet aliases defined in a GameLift region.
• **DescribeAlias** – Retrieve information on an existing alias.
• **UpdateAlias** – Change settings for an alias, such as redirecting it from one fleet to another.
• **DeleteAlias** – Remove an alias from the region.
• **ResolveAlias** – Get the fleet ID that a specified alias points to.

**Manage VPC peering connections for fleets.** Use VPC peering to establish secure access between your GameLift and other AWS resources.
• **CreateVpcPeeringAuthorization** – Authorize a peering connection to one of your VPCs.
• **DescribeVpcPeeringAuthorizations** – Retrieve valid peering connection authorizations.
• **DeleteVpcPeeringAuthorization** – Delete a peering connection authorization.
• **CreateVpcPeeringConnection** – Establish a peering connection between the VPC for a GameLift fleet and one of your VPCs.
• **DescribeVpcPeeringConnections** – Retrieve information on active or pending VPC peering connections with a GameLift fleet.
• **DeleteVpcPeeringConnection** – Delete a VPC peering connection with a GameLift fleet.

**APIs for GameLift FleetIQ**

These APIs are part of the standalone GameLift FleetIQ solution. Use these operations to manage game server groups and game servers that are being hosted with Amazon EC2 resources in conjunction with Auto Scaling groups. Learn more about standalone GameLift FleetIQ in [GameLift FleetIQ Guide [PUBLIC PREVIEW]](p. 191).

• **Manage game server groups.** Configure a game server group to deploy game servers on Amazon EC2 resources and optimize hosting for games.
  • **CreateGameServerGroup** – Create a new game server group, set up a corresponding Auto Scaling group, and begin launching instances to host your game server.
  • **ListGameServerGroups** – Get a list of all game server groups in a GameLift region.
  • **DescribeGameServerGroup** – Retrieve metadata for a game server group.
  • **UpdateGameServerGroup** – Change game server group metadata.
  • **DeleteGameServerGroup** – Permanently remove a game server group and terminates FleetIQ activity for the associated hosting resources.
  • **ResumeGameServerGroup** – Reinstate suspended FleetIQ activity for a game server group.
  • **SuspendGameServerGroup** – Temporarily stop FleetIQ activity for a game server group.

• **Manage game server status.** Track game server status and respond to requests for new game sessions.
  • **RegisterGameServer** – Notify GameLift FleetIQ that a new game server is ready to host gameplay.
  • **ListGameServers** – Get a list of all game servers that are currently running in a game server group.
  • **ClaimGameServer** – Locate and reserve a game server to host a new game session.
  • **DescribeGameServer** – Retrieve metadata for a game server.
  • **UpdateGameServer** – Change game server metadata, health status, or utilization status.
  • **DeregisterGameServer** – Remove a game server resource from the game server group.

**Available Programming Languages**

The AWS SDK with Amazon GameLift is available in the following languages. See documentation for each language for details on support for development environments.
Amazon GameLift Realtime Servers Reference

This section contains reference documentation for the Amazon GameLift Realtime Servers SDK. It includes the Realtime Client API as well as guidance for configuring your Realtime Servers script.

Topics
- Realtime Servers Client API (C#) Reference (p. 217)
- Amazon GameLift Realtime Servers Script Reference (p. 227)

Realtime Servers Client API (C#) Reference

Use the Realtime Client API to prepare your multiplayer game clients for use with Amazon GameLift Realtime Servers. For more on the integration process, see Get Started with Realtime Servers (p. 31). The Client API contains a set of synchronous API calls and asynchronous callbacks that enable a game client to connect to a Realtime server and exchange messages and data with other game clients via the server.

This API is defined in the following libraries:

Client.cs
- Synchronous Actions (p. 217)
- Asynchronous Callbacks (p. 221)
- Data Types (p. 223)

To set up the Realtime Client API

1. Download the Amazon GameLift Realtime Client SDK.
2. Build the C# SDK libraries. Locate the solution file GameScaleLightweightClientSdkNet45.sln. See the README.md file for the C# Server SDK for minimum requirements and additional build options. In an IDE, load the solution file. To generate the SDK libraries, restore the NuGet packages and build the solution.
3. Add the Realtime Client libraries to your game client project.

Realtime Servers Client API (C#) Reference: Actions

This C# Realtime Client API reference can help you prepare your multiplayer game for use with Realtime Servers deployed on Amazon GameLift fleets. For details on the integration process, see Get Started with Realtime Servers (p. 31).

- Synchronous Actions
- Asynchronous Callbacks (p. 221)
• Data Types (p. 223)

Client()

Initializes a new client to communicate with the Realtime server and identifies the type of connection to use.

Syntax

```csharp
public Client(ClientConfiguration configuration)
```

Parameters

clientConfiguration

Configuration details specifying the client/server connection type. You can opt to call Client() without this parameter; however, this approach results in an unsecured connection by default.

Type: ClientConfiguration (p. 224)

Required: No

Return Value

Returns an instance of the Realtime client for use with communicating with the Realtime server.

Connect()

Requests a connection to a server process that is hosting a game session.

Syntax

```csharp
public ConnectionStatus Connect(string endpoint, int remoteTcpPort, int listenPort, ConnectionToken token)
```

Parameters

endpoint

DNS name or IP address of the game session to connect to. The endpoint is specified in a GameSession object, which is returned in response to a client call to the AWS SDK Amazon GameLift API actions StartGameSessionPlacement, CreateGameSession, or DescribeGameSessions.

**Note**

If the Realtime server is running on a fleet with a TLS certificate, you must use the DNS name.

Type: String

Required: Yes

remoteTcpPort

Port number for the TCP connection assigned to the game session. This information is specified in a GameSession object, which is returned in response to a StartGameSessionPlacement CreateGameSession, or DescribeGameSessions request.

Type: Integer
**Valid Values**: 1900 to 2000.
**Required**: Yes

**listenPort**
Port number that the game client is listening on for messages sent using the UDP channel.
**Type**: Integer
**Valid Values**: 33400 to 33500.
**Required**: Yes

**token**
Optional information that identifies the requesting game client to the server process.
**Type**: ConnectionToken (p. 224)
**Required**: Yes

**Return Value**
Returns a ConnectionStatus (p. 226) enum value indicating the client's connection status.

**Disconnect()**
When connected to a game session, disconnects the game client from the game session.

**Syntax**

```csharp
public void Disconnect()
```

**Parameters**
This action has no parameters.

**Return Value**
This method does not return anything.

**NewMessage()**
Creates a new message object with a specified operation code. Once a message object is returned, complete the message content by specifying a target, updating the delivery method, and adding a data payload as needed. Once completed, send the message using SendMessage().

**Syntax**

```csharp
public RTMessage NewMessage(int opCode)
```

**Parameters**

**opCode**
Developer-defined operation code that identifies a game event or action, such as a player move or a server notification.
**Type**: Integer
**Required**: Yes
Return Value

Returns an RTMessage (p. 224) object containing the specified operation code and default delivery method. The delivery intent parameter is set to FAST by default.

SendMessage()

Sends a message to a player or group using the delivery method specified.

Syntax

```csharp
public void SendMessage(RTMessage message)
```

Parameters

- **message**
  
  Message object that specifies the target recipient, delivery method, and message content.
  
  Type: RTMessage (p. 224)
  
  Required: Yes

Return Value

This method does not return anything.

JoinGroup()

Adds the player to the membership of a specified group. Groups can contain any of the players that are connected to the game. Once joined, the player receives all future messages sent to the group and can send messages to the entire group.

Syntax

```csharp
public void JoinGroup(int targetGroup)
```

Parameters

- **targetGroup**
  
  Unique ID that identifies the group to add the player to. Group IDs are developer-defined.
  
  Type: Integer
  
  Required: Yes

Return Value

This method does not return anything. Because this request is sent using the reliable (TCP) delivery method, a failed request triggers the callback OnError() (p. 222).

LeaveGroup()

Removes the player from the membership of a specified group. Once no longer in the group, the player does not receive messages sent to the group and cannot send messages to the entire group.
**Syntax**

```csharp
public void LeaveGroup(int targetGroup)
```

**Parameters**

**targetGroup**

Unique ID identifying the group to remove the player from. Group IDs are developer-defined.

Type: Integer

Required: Yes

**Return Value**

This method does not return anything. Because this request is sent using the reliable (TCP) delivery method, a failed request triggers the callback `OnError()` (p. 222).

**RequestGroupMembership()**

Requests that a list of players in the specified group be sent to the game client. Any player can request this information, regardless of whether they are a member of the group or not. In response to this request, the membership list is sent to the client via an `OnGroupMembershipUpdated()` (p. 223) callback.

**Syntax**

```csharp
public void RequestGroupMembership(int targetGroup)
```

**Parameters**

**targetGroup**

Unique ID identifying the group to get membership information for. Group IDs are developer-defined.

Type: Integer

Required: Yes

**Return Value**

This method does not return anything.

**Realtime Servers Client API (C#) Reference: Asynchronous Callbacks**

Use this C# Realtime Client API reference to help you prepare your multiplayer game for use with Realtime Servers deployed on Amazon GameLift fleets. For details on the integration process, see Get Started with Realtime Servers (p. 31).

- Synchronous Actions (p. 217)
- Asynchronous Callbacks
- Data Types (p. 223)
A game client needs to implement these callback methods to respond to events. The Realtime server invokes these callbacks to send game-related information to the game client. Callbacks for the same events can also be implemented with custom game logic in the Realtime server script. See Script Callbacks for Realtime Servers (p. 227).

Callback methods are defined in `ClientEvents.cs`.

**OnOpen()**

Invoked when the server process accepts the game client's connection request and opens a connection.

**Syntax**

```csharp
public void OnOpen()
```

**Parameters**

This method takes no parameters.

**Return Value**

This method does not return anything.

**OnClose()**

Invoked when the server process terminates the connection with the game client, such as after a game session ends.

**Syntax**

```csharp
public void OnClose()
```

**Parameters**

This method takes no parameters.

**Return Value**

This method does not return anything.

**OnError()**

Invoked when a failure occurs for a Realtime Client API request. This callback can be customized to handle a variety of connection errors.

**Syntax**

```csharp
private void OnError(byte[] args)
```

**Parameters**

This method takes no parameters.

**Return Value**

This method does not return anything.
OnDataReceived()

Invoked when the game client receives a message from the Realtime server. This is the primary method by which messages and notifications are received by a game client.

Syntax

```csharp
public void OnDataReceived(DataReceivedEventArgs dataReceivedEventArgs)
```

Parameters

dataReceivedEventArgs

Information related to message activity.

Type: DataReceivedEventArgs (p. 225)

Required: Yes

Return Value

This method does not return anything.

OnGroupMembershipUpdated()

Invoked when the membership for a group that the player belongs to has been updated. This callback is also invoked when a client calls RequestGroupMembership.

Syntax

```csharp
public void OnGroupMembershipUpdated(GroupMembershipEventArgs groupMembershipEventArgs)
```

Parameters

groupMembershipEventArgs

Information related to group membership activity.

Type: GroupMembershipEventArgs (p. 226)

Required: Yes

Return Value

This method does not return anything.

Realtime Servers Client API (C#) Reference: Data Types

This C# Realtime Client API reference can help you prepare your multiplayer game for use with Realtime Servers deployed on Amazon GameLift fleets. For details on the integration process, see Get Started with Realtime Servers (p. 31).

- Synchronous Actions (p. 217)
- Asynchronous Callbacks (p. 221)
- Data Types
ClientConfiguration

Information about how the game client connects to a Realtime server.

Contents

ConnectionType

Type of client/server connection to use, either secured or unsecured. If you don't specify a connection type, the default is unsecured.

Note
When connecting to a Realtime server on a secured fleet with a TLS certificate, you must use the value RT_OVER_WSS_DTLS_TLS12.

Type: A ConnectionType enum (p. 226) value.

Required: No

ConnectionToken

Information about the game client and/or player that is requesting a connection with a Realtime server.

Contents

playerSessionId

Unique ID issued by GameLift when a new player session is created. A player session ID is specified in a PlayerSession object, which is returned in response to a client call to the AWS SDK Amazon GameLift API actions StartGameSessionPlacement, CreateGameSession, DescribeGameSessionPlacement, or DescribePlayerSessions.

Type: String

Required: No

payload

Developer-defined information to be communicated to the Realtime server on connection. This includes any arbitrary data that might be used for a custom sign-in mechanism. For examples, a payload may provide authentication information to be processed by the Realtime server script before allowing a client to connect.

Type: byte array

Required: No

RTMessage

Content and delivery information for a message. A message must specify either a target player or a target group.

Contents

opCode

Developer-defined operation code that identifies a game event or action, such as a player move or a server notification. A message's Op code provides context for the data payload that is being provided. Messages that are created using NewMessage() already have the operation code set, but it can be changed at any time.
**DataReceivedEventArgs**

Data provided with an [OnDataReceived()](p. 223) callback.

**Contents**

**sender**

Unique ID identifying the entity (player ID or server ID) who originated the message.

Type: Integer

Required: Yes

**opCode**

Developer-defined operation code that identifies a game event or action, such as a player move or a server notification. A message's op code provides context for the data payload that is being provided.

Type: Integer

Required: Yes
data
Message content. This information is structured as needed to be processed by the game client based on the accompanying operation code. It may contain game state data or other information that needs to be communicated between game clients or between a game client and the Realtime server.

Type: Byte array
Required: No

GroupMembershipEventArgs
Data provided with an OnGroupMembershipUpdated() (p. 223) callback.

Contents

sender
Unique ID identifying the player who requested a group membership update.

Type: Integer
Required: Yes

opCode
Developer-defined operation code that identifies a game event or action.

Type: Integer
Required: Yes

groupId
Unique ID identifying the group that is the intended recipient of the message being sent. Group IDs are developer defined.

Type: Integer
Required: Yes

playerId
List of player IDs who are current members of the specified group.

Type: Integer array
Required: Yes

Enums

 Enums defined for the Realtime Client SDK are defined as follows:

ConnectionStatus
- CONNECTED – Game client is connected to the Realtime server with a TCP connection only. All messages regardless of delivery intent are sent via TCP.
- CONNECTED_SEND_FAST – Game client is connected to the Realtime server with a TCP and a UDP connection. However, the ability to receive messages via UDP is not yet verified; as a result, all messages sent to the game client use TCP.
- CONNECTED_SEND_AND_RECEIVE_FAST – Game client is connected to the Realtime server with a TCP and a UDP connection. The game client can send and receive messages using either TCP or UDP.
• **CONNECTING** Game client has sent a connection request and the Realtime server is processing it.
• **DISCONNECTED_CLIENT_CALL** – Game client was disconnected from the Realtime server in response to a `Disconnect()` (p. 219) request from the game client.
• **DISCONNECTED** – Game client was disconnected from the Realtime server for a reason other than a client disconnect call.

**ConnectionType**

• **RT_OVER_WSS_DTLS_TLS12** – Secure connection type.
  
  For use with Realtime servers that are running on a GameLift fleet with a TLS certificate generated. When using a secure connection, TCP traffic is encrypted using TLS 1.2, and UDP traffic is encrypted using DTLS 1.2.
• **RT_OVER_WS_UDP_UNSECURED** – Non-secure connection type.
• **RT_OVER_WEBSOCKET** – Non-secure connection type. This value is no longer preferred.

**DeliveryIntent**

• **FAST** – Delivered using a UDP channel.
• **RELIABLE** – Delivered using a TCP connection.

---

**Amazon GameLift Realtime Servers Script Reference**

Use these resources to build out custom logic in your Realtime scripts.

**Topics**

• [Script Callbacks for Realtime Servers (p. 227)](#)
• [Realtime Servers Interface (p. 229)](#)

**Script Callbacks for Realtime Servers**

You can provide custom logic to respond to events by implementing these callbacks in your Realtime script.

**init**

Initializes the Realtime server and receives a Realtime server interface.

**Syntax**

```
init(rtsession)
```

**onMessage**

Invoked when a received message is sent to the server.

**Syntax**

```
onMessage(gameMessage)
```

**onHealthCheck**

Invoked to set the status of the game session health. By default, health status is healthy (or `true`). This callback can be implemented to perform custom health checks and return a status.
Syntax

onHealthCheck()

onStartGameSession
Invoked when a new game session starts, with a game session object passed in.
Syntax

onStartGameSession(session)

onProcessTerminate
Invoked when the server process is being terminated by the Amazon GameLift service. This can act as a trigger to exit cleanly from the game session. There is no need to call processEnding().
Syntax

onProcessTerminate()

onPlayerConnect
Invoked when a player requests a connection and has passed initial validation.
Syntax

onPlayerConnect(connectMessage)

onPlayerAccepted
Invoked when a player connection is accepted.
Syntax

onPlayerAccepted(player)

onPlayerDisconnect
Invoked when a player disconnects from the game session, either by sending a disconnect request or by other means.
Syntax

onPlayerDisconnect(peerId)

onProcessStarted
Invoked when a server process is started. This callback allows the script to perform any custom tasks needed to prepare to host a game session.
Syntax

onProcessStarted(args)
onSendToPlayer

Invoked when a message is received on the server from one player to be delivered to another player. This process runs before the message is delivered.

Syntax

```javascript
onSendToPlayer(gameMessage)
```

onSendToGroup

Invoked when a message is received on the server from one player to be delivered to a group. This process runs before the message is delivered.

Syntax

```javascript
onSendToGroup(gameMessage)
```

onPlayerJoinGroup

Invoked when a player sends a request to join a group.

Syntax

```javascript
onPlayerJoinGroup(groupId, peerId)
```

onPlayerLeaveGroup

Invoked when a player sends a request to leave a group.

Syntax

```javascript
onPlayerLeaveGroup(groupId, peerId)
```

Realtime Servers Interface

When a Realtime script initializes, an interface to the Realtime server is returned. This topic describes the properties and methods available through the interface. Learn more about writing Realtime scripts and view a detailed script example in Creating a Realtime Script (p. 63).

The Realtime interface provides access to the following objects:

- session
- player
- gameMessage

Realtime Session object

Use these methods to access server-related information and perform server-related actions.

getPlayers()

Retrieves a list of peer IDs for players that are currently connected to the game session. Returns an array of player objects.
Syntax

```javascript
rtSession.getPlayers()
```

**broadcastGroupMembershipUpdate()**

Triggers delivery of an updated group membership list to player group. Specify which membership to broadcast (groupIdToBroadcast) and the group to receive the update (targetGroupId).

Syntax

```javascript
rtSession.broadcastGroupMembershipUpdate(groupIdToBroadcast, targetGroupId)
```

**getServerId()**

Retrieves the server's unique peer ID identifier, which is used to route messages to the server.

Syntax

```javascript
rtSession.getServerId()
```

**getAllPlayersGroupId()**

Retrieves the group ID for the default group that contains all players currently connected to the game session.

Syntax

```javascript
rtSession.getAllPlayersGroupId()
```

**processEnding()**

Triggers the Realtime server to terminate the game server. This function must be called from the Realtime script to exit cleanly from a game session.

Syntax

```javascript
rtSession.processEnding()
```

**getGameSessionId()**

Retrieves the unique ID of the game session currently running.

Syntax

```javascript
rtSession.getGameSessionId()
```

**getLogger()**

Retrieves the interface for logging. Use this to log statements that will be captured in your game session logs. The logger supports use of "info", "warn", and "error" statements. For example:

```javascript
logger.info("<string>").
```
Syntax

```javascript
rtSession.getLogger()
```

**sendMessage()**

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to a player recipient using the UDP channel. Identify the recipient using the player's peer ID.

**Syntax**

```javascript
rtSession.sendMessage(gameMessage, targetPlayer)
```

**sendGroupMessage()**

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to all players in a player group using the UDP channel. Identify the recipients using the group ID.

**Syntax**

```javascript
rtSession.sendGroupMessage(gameMessage, targetGroup)
```

**sendReliableMessage()**

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to a player recipient using the TCP channel. Identify the recipient using the player's peer ID.

**Syntax**

```javascript
rtSession.sendReliableMessage(gameMessage, targetPlayer)
```

**sendReliableGroupMessage()**

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to all players in a player group using the TCP channel. Identify the recipients using the group ID.

**Syntax**

```javascript
rtSession.sendReliableGroupMessage(gameMessage, targetGroup)
```

**newTextGameMessage()**

Creates a new message containing text, to be sent from the server to player recipients using the `sendMessage` functions. Message format is similar to the format used in the Realtime Client SDK (see `RTMessage (p. 224)`). Returns a `gameMessage` object.

**Syntax**

```javascript
rtSession.newTextGameMessage(opcode, sender, payload)
```

**newBinaryGameMessage()**

Creates a new message containing binary data, to be sent from the server to player recipients using the `sendMessage` functions. Message format is similar to the format used in the Realtime Client SDK (see `RTMessage (p. 224)`). Returns a `gameMessage` object.
Syntax

```plaintext
rtSession.newBinaryGameMessage(opcode, sender, binaryPayload)
```

**Player object**

Access player-related information.

```plaintext
player.peerId
```

Unique ID that is assigned to a game client when it connects to the Realtime server and joined the game session.

```plaintext
player.playerSessionId
```

Player session ID that was referenced by the game client when it connected to the Realtime server and joined the game session.

**Game message object**

Use these methods to access messages that are received by the Realtime server. Messages received from game clients have the `RTMessage` (p. 224) structure.

```plaintext
gameMessage.opcode
```

Operation code contained in a message.

```plaintext
gameMessage.payload
```

Payload contained in a message. May be text or binary.

```plaintext
gameMessage.sender
```

Peer ID of the game client that sent a message.

```plaintext
gameMessage.reliable
```

Boolean indicating whether the message was sent via TCP (true) or UDP (false).

---

**Amazon GameLift Server SDK Reference**

This section contains reference documentation for the Amazon GameLift Server SDK. Use the Server SDK to integrate your custom game server with the Amazon GameLift service to start and manage game servers as needed.

**Topics**

- Amazon GameLift Server API (C++) Reference (p. 232)
- Amazon GameLift Server API (C#) Reference (p. 248)
- Amazon GameLift Server API Reference for Unreal Engine (p. 261)

**Amazon GameLift Server API (C++) Reference**

This Amazon GameLift C++ Server API reference can help you prepare your multiplayer game for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).
This API is defined in GameLiftServerAPI.h, LogParameters.h, and ProcessParameters.h.

- Actions (p. 233)
- Data Types (p. 244)

Amazon GameLift Server API (C++) Reference: Actions

This Amazon GameLift C++ Server API reference can help you prepare your multiplayer game for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).

This API is defined in GameLiftServerAPI.h, LogParameters.h, ProcessParameters.h.

- Actions
- Data Types (p. 244)

AcceptPlayerSession()

Notifies the Amazon GameLift service that a player with the specified player session ID has connected to the server process and needs validation. Amazon GameLift verifies that the player session ID is valid—that is, that the player ID has reserved a player slot in the game session. Once validated, Amazon GameLift changes the status of the player slot from RESERVED to ACTIVE.

Syntax

```cpp
GenericOutcome AcceptPlayerSession(const std::string& playerSessionId);
```

Parameters

playerSessionId

Unique ID issued by the Amazon GameLift service in response to a call to the AWS SDK Amazon GameLift API action `CreatePlayerSession`. The game client references this ID when connecting to the server process.

Type: std::string

Required: No

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates a function for handling a connection request, including validating and rejecting invalid player session IDs.

```cpp
void ReceiveConnectingPlayerSessionID (Connection& connection, const std::string& playerSessionId){
    Aws::GameLift::GenericOutcome connectOutcome =
        Aws::GameLift::Server::AcceptPlayerSession(playerSessionId);
    if(connectOutcome.IsSuccess())
    {
        connectionToSessionMap.emplace(connection, playerSessionId);
    }
```
connection.Accept();
} else {
    connection.Reject(connectOutcome.GetError().GetMessage());
}

**ActivateGameSession()**

Notifies the Amazon GameLift service that the server process has started a game session and is now ready to receive player connections. This action should be called as part of the `onStartGameSession()` callback function, after all game session initialization has been completed.

**Syntax**

```
GenericOutcome ActivateGameSession();
```

**Parameters**

This action has no parameters.

**Return Value**

Returns a generic outcome consisting of success or failure with an error message.

**Example**

This example shows `ActivateGameSession()` being called as part of the `onStartGameSession()` callback function.

```
void onStartGameSession(Aws::GameLift::Model::GameSession myGameSession)
{
    // game-specific tasks when starting a new game session, such as loading map
    GenericOutcome outcome = Aws::GameLift::Server::ActivateGameSession();
}
```

**DescribePlayerSessions()**

Retrieves player session data, including settings, session metadata, and player data. Use this action to get information for a single player session, for all player sessions in a game session, or for all player sessions associated with a single player ID.

**Syntax**

```
DescribePlayerSessionsOutcome DescribePlayerSessions (const Aws::GameLift::Server::Model::DescribePlayerSessionsRequest &describePlayerSessionsRequest);
```

**Parameters**

**describePlayerSessionsRequest**

A `DescribePlayerSessionsRequest (p. 244)` object describing which player sessions to retrieve.

Required: Yes
Return Value

If successful, returns a DescribePlayerSessionsOutcome object containing a set of player session objects that fit the request parameters. Player session objects have a structure identical to the AWS SDK Amazon GameLift API PlayerSession data type.

Example

This example illustrates a request for all player sessions actively connected to a specified game session. By omitting NextToken and setting the Limit value to 10, Amazon GameLift returns the first 10 player sessions records matching the request.

```cpp
// Set request parameters
Aws::GameLift::Server::Model::DescribePlayerSessionsRequest request;
request.SetPlayerSessionStatusFilter(Aws::GameLift::Server::Model::PlayerSessionStatusMapper::GetNameForPlayerSessionStatus(Aws::GameLift::Server::Model::PlayerSessionStatus::Active));
request.SetLimit(10);
request.SetGameSessionId("the game session ID");    // can use GetGameSessionId()

// Call DescribePlayerSessions
Aws::GameLift::DescribePlayerSessionsOutcome playerSessionsOutcome =
    Aws::GameLift::Server::DescribePlayerSessions(request);
```

GetGameSessionId()

Retrieves a unique identifier for the game session currently being hosted by the server process, if the server process is active. The identifier is returned in ARN format: arn:aws:gamelift:<region>::gamesession/fleet-<fleet ID>/<ID string>.

Syntax

```cpp
AwsStringOutcome GetGameSessionId();
```

Parameters

This action has no parameters.

Return Value

If successful, returns the game session ID as an AwsStringOutcome object. If not successful, returns an error message.

Example

```cpp
Aws::GameLift::AwsStringOutcome sessionIdOutcome =
    Aws::GameLift::Server::GetGameSessionId();
```

GetInstanceCertificate()

Retrieves the file location of a pem-encoded TLS certificate that is associated with the fleet and its instances. This certificate is generated when a new fleet is created with the certificate configuration set to GENERATED. Use this certificate to establish a secure connection with a game client and to encrypt client/server communication.

Syntax

```cpp
GetInstanceCertificateOutcome GetInstanceCertificate();
```
Parameters
This action has no parameters.

Return Value
If successful, returns a GetInstanceCertificateOutcome object containing the location of the fleet's TLS certificate file, which is stored on the instance. If not successful, returns an error message.

Example

```cpp
Aws::GameLift::GetInstanceCertificateOutcome certificateOutcome =
    Aws::GameLift::Server::GetInstanceCertificate();
```

GetSdkVersion()

Returns the current version number of the SDK in use.

Syntax

```cpp
AwsStringOutcome GetSdkVersion();
```

Parameters
This action has no parameters.

Return Value
If successful, returns the current SDK version as an AwsStringOutcome object. The returned string includes the version number only (ex. "3.1.5"). If not successful, returns an error message.

Example

```cpp
Aws::GameLift::AwsStringOutcome SdkVersionOutcome =
    Aws::GameLift::Server::GetSdkVersion();
```

GetTerminationTime()

Returns the time that a server process is scheduled to be shut down, if a termination time is available. A server process takes this action after receiving an onProcessTerminate() callback from the Amazon GameLift service. A server process may be shut down for several reasons: (1) process poor health, (2) when an instance is being terminated during a scale-down event, or (3) when an instance is being terminated due to a spot-instance interruption (p. 103).

If the process has received an onProcessTerminate() callback, the value returned is the estimated termination time in epoch seconds. If no termination time is available, the value returned is -1, which indicates that the server process may be terminated at any time. If the process has not received an onProcessTerminate() callback, the returned value will always be -1. Learn more about shutting down a server process (p. 45).

Syntax

```cpp
AwsLongOutcome GetTerminationTime();
```

Parameters
This action has no parameters.
Return Value

If successful, returns the current SDK version as an `AwsLongOutcome` object. The value is either the termination time in epoch seconds, or the value -1. If not successful, returns an error message.

Example

```cpp
Aws::GameLift::AwsLongOutcome TermTimeOutcome = 
    Aws::GameLift::Server::GetTerminationTime();
```

**InitSDK()**

Initializes the Amazon GameLift SDK. This method should be called on launch, before any other Amazon GameLift-related initialization occurs.

**Syntax**

```cpp
InitSDKOutcome InitSDK();
```

**Parameters**

This action has no parameters.

**Return Value**

If successful, returns an `InitSdkOutcome` object indicating that the server process is ready to call `ProcessReady()` (p. 238).

Example

```cpp
Aws::GameLift::Server::InitSDKOutcome initOutcome = 
    Aws::GameLift::Server::InitSDK();
```

**ProcessEnding()**

Notifies the Amazon GameLift service that the server process is shutting down. This method should exit with an exit code of 0; a non-zero exit code results in an event message that the process did not exit cleanly.

**Syntax**

```cpp
GenericOutcome ProcessEnding();
```

**Parameters**

This action has no parameters.

**Return Value**

Returns a generic outcome consisting of success or failure with an error message.

Example

```cpp
Aws::GameLift::GenericOutcome outcome = Aws::GameLift::Server::ProcessEnding();
```
ProcessReady()

Notifies the Amazon GameLift service that the server process is ready to host game sessions. This method should be called after successfully invoking InitSDK() (p. 237) and completing any setup tasks required before the server process can host a game session.

This call is synchronous. To make an asynchronous call, use ProcessReadyAsync() (p. 239). See Prepare a Server Process (p. 42) for more details.

Syntax

```cpp
GenericOutcome ProcessReady(
    const Aws::GameLift::Server::ProcessParameters &processParameters);
```

Parameters

**processParameters**

A ProcessParameters (p. 245) object communicating the following information about the server process:

- Names of callback methods, implemented in the game server code, that the Amazon GameLift service invokes to communicate with the server process.
- Port number that the server process is listening on.
- Path to any game session-specific files that you want Amazon GameLift to capture and store.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates both the ProcessReady() (p. 238) call and callback function implementations.

```cpp
// Set parameters and call ProcessReady
std::string serverLog("serverOut.log");        // Example of a log file written by the game
server
std::vector<std::string> logPaths;
logPaths.push_back(serverLog);

int listenPort = 9399;

Aws::GameLift::Server::ProcessParameters processReadyParameter =
    Aws::GameLift::Server::ProcessParameters(
        std::bind(&Server::onStartGameSession, this, std::placeholders::_1),
        std::bind(&Server::onProcessTerminate, this),
        std::bind(&Server::OnHealthCheck, this),
        std::bind(&Server::OnUpdateGameSession, this),
        listenPort,
        Aws::GameLift::Server::LogParameters(logPaths));

Aws::GameLift::GenericOutcome outcome =
    Aws::GameLift::Server::ProcessReady(processReadyParameter);

// Implement callback functions
void Server::onStartGameSession(Aws::GameLift::Model::GameSession myGameSession) {
```
// game-specific tasks when starting a new game session, such as loading map
GenericOutcome outcome = Aws::GameLift::Server::ActivateGameSession (maxPlayers);
}

void Server::onProcessTerminate()
{
    // game-specific tasks required to gracefully shut down a game session,
    // such as notifying players, preserving game state data, and other cleanup
    GenericOutcome outcome = Aws::GameLift::Server::ProcessEnding();
}

bool Server::onHealthCheck()
{
    bool health;
    // complete health evaluation within 60 seconds and set health
    return health;
}

**ProcessReadyAsync()**

Notifies the Amazon GameLift service that the server process is ready to host game sessions. This method should be called once the server process is ready to host a game session. The parameters specify the names of callback functions for Amazon GameLift to call in certain circumstances. Game server code must implement these functions.

This call is asynchronous. To make a synchronous call, use `ProcessReady()` (p. 238). See Prepare a Server Process (p. 42) for more details.

**Syntax**

```
GenericOutcomeCallable ProcessReadyAsync(
    const Aws::GameLift::Server::ProcessParameters &processParameters);
```

**Parameters**

**processParameters**

A `ProcessParameters` (p. 245) object communicating the following information about the server process:

- Names of callback methods, implemented in the game server code, that the Amazon GameLift service invokes to communicate with the server process.
- Port number that the server process is listening on.
- Path to any game session-specific files that you want Amazon GameLift to capture and store.

Required: Yes

**Return Value**

Returns a generic outcome consisting of success or failure with an error message.

**Example**

```
// Set parameters and call ProcessReady
std::string serverLog("serverOut.log");        // This is an example of a log file written
by the game server
std::vector<std::string> logPaths;
logPaths.push_back(serverLog);
```
int listenPort = 9339;

Aws::GameLift::Server::ProcessParameters processReadyParameter =
    Aws::GameLift::Server::ProcessParameters(
        std::bind(&Server::onStartGameSession, this, std::placeholders::_1),
        std::bind(&Server::onProcessTerminate, this),
        std::bind(&Server::OnHealthCheck, this),
        std::bind(&Server::OnUpdateGameSession, this),
        listenPort,
        Aws::GameLift::Server::LogParameters(logPaths));

Aws::GameLift::GenericOutcomeCallable outcome =
    Aws::GameLift::Server::ProcessReadyAsync(processReadyParameter);

// Implement callback functions
void onStartGameSession(Aws::GameLift::Model::GameSession myGameSession)
{
    // game-specific tasks when starting a new game session, such as loading map
    GenericOutcome outcome = Aws::GameLift::Server::ActivateGameSession (maxPlayers);
}

void onProcessTerminate()
{
    // game-specific tasks required to gracefully shut down a game session,
    // such as notifying players, preserving game state data, and other cleanup
    GenericOutcome outcome = Aws::GameLift::Server::ProcessEnding();
}

bool onHealthCheck()
{
    // perform health evaluation and complete within 60 seconds
    return health;
}

RemovePlayerSession()

Notifies the Amazon GameLift service that a player with the specified player session ID has disconnected
from the server process. In response, Amazon GameLift changes the player slot to available, which allows
it to be assigned to a new player.

Syntax

GenericOutcome RemovePlayerSession(
    const std::string& playerSessionId);

Parameters

playerSessionId

Unique ID issued by the Amazon GameLift service in response to a call to the AWS SDK Amazon
GameLift API action CreatePlayerSession. The game client references this ID when connecting to the
server process.

Type: std::string

Required: No

Return Value

Returns a generic outcome consisting of success or failure with an error message.
Example

```cpp
Aws::GameLift::GenericOutcome disconnectOutcome = 
    Aws::GameLift::Server::RemovePlayerSession(playerSessionId);
```

StartMatchBackfill()

Sends a request to find new players for open slots in a game session created with FlexMatch. See also the AWS SDK action `StartMatchBackfill()`. With this action, match backfill requests can be initiated by a game server process that is hosting the game session. Learn more about the FlexMatch backfill feature in Backfill Existing Games with FlexMatch (p. 76).

This action is asynchronous. If new players are successfully matched, the Amazon GameLift service delivers updated matchmaking data using the callback function `OnUpdateGameSession()`.

A server process can have only one active match backfill request at a time. To send a new request, first call `StopMatchBackfill()` (p. 242) to cancel the original request.

Syntax

```cpp
StartMatchBackfillOutcome StartMatchBackfill(
    const Aws::GameLift::Server::Model::StartMatchBackfillRequest &startBackfillRequest);
```

Parameters

**StartMatchBackfillRequest**

A `StartMatchBackfillRequest` (p. 246) object that communicates the following information:

- A ticket ID to assign to the backfill request. This information is optional; if no ID is provided, Amazon GameLift will autogenerate one.
- The matchmaker to send the request to. The full configuration ARN is required. This value can be acquired from the game session's matchmaker data.
- The ID of the game session that is being backfilled.
- Available matchmaking data for the game session's current players.

Required: Yes

Return Value

Returns a `StartMatchBackfillOutcome` object with the match backfill ticket or failure with an error message. Ticket status can be tracked using the AWS SDK action `DescribeMatchmaking()`.

Example

```cpp
// Build a backfill request
std::vector<Player> players;
Aws::GameLift::Server::Model::StartMatchBackfillRequest startBackfillRequest;
startBackfillRequest.SetTicketId("a ticket ID"); // optional, autogenerated if not provided
startBackfillRequest.SetMatchmakingConfigurationArn("the matchmaker configuration ARN"); // from the game session matchmaker data
startBackfillRequest.SetGameSessionArn("the game session ARN"); // can use GetGameSessionId()
startBackfillRequest.SetPlayers(players); //from the game session matchmaker data

// Send backfill request
Aws::GameLift::StartMatchBackfillOutcome backfillOutcome =
```
StopMatchBackfill()

Cancels an active match backfill request that was created with StartMatchBackfill() (p. 241). See also the AWS SDK action StopMatchmaking(). Learn more about the FlexMatch backfill feature in Backfill Existing Games with FlexMatch (p. 76).

Syntax

```cpp
GenericOutcome StopMatchBackfill(
    const Aws::GameLift::Server::Model::StopMatchBackfillRequest &stopBackfillRequest);
```

Parameters

StopMatchBackfillRequest

A StopMatchBackfillRequest (p. 247) object identifying the matchmaking ticket to cancel:
- ticket ID assigned to the backfill request being canceled
- matchmaker the backfill request was sent to
- game session associated with the backfill request

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

```cpp
// Set backfill stop request parameters
Aws::GameLift::Server::Model::StopMatchBackfillRequest stopBackfillRequest;
stopBackfillRequest.SetTicketId("the ticket ID");
stopBackfillRequest.SetGameSessionArn("the game session ARN"); // can use GetGameSessionId()
stopBackfillRequest.SetMatchmakingConfigurationArn("the matchmaking configuration ARN"); // from the game session matchmaker data

Aws::GameLift::GenericOutcome stopBackfillOutcome = 
    Aws::GameLift::Server::StopMatchBackfillRequest(stopBackfillRequest);
```

TerminateGameSession()

Notifies the Amazon GameLift service that the server process has shut down the game session. Since each server process hosts only one game session at a time, there's no need to specify which session. This action should be called at the end of the game session shutdown process. After calling this action, the server process can call ProcessReady() (p. 238) to signal its availability to host a new game session. Alternatively it can call ProcessEnding() (p. 237) to shut down the server process and terminate the instance.
Syntax

```
GenericOutcome TerminateGameSession();
```

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates a server process at the end of a game session.

```cpp
// game-specific tasks required to gracefully shut down a game session,
// such as notifying players, preserving game state data, and other cleanup
Aws::GameLift::GenericOutcome outcome =
    Aws::GameLift::Server::TerminateGameSession();
Aws::GameLift::GenericOutcome outcome =
    Aws::GameLift::Server::ProcessReady(onStartGameSession, onProcessTerminate);
```

**UpdatePlayerSessionCreationPolicy()**

Updates the current game session's ability to accept new player sessions. A game session can be set to either accept or deny all new player sessions. See also the AWS SDK action `UpdateGameSession()`.

Syntax

```
GenericOutcome UpdatePlayerSessionCreationPolicy(
    Aws::GameLift::Model::PlayerSessionCreationPolicy newPlayerSessionPolicy);
```

Parameters

**newPlayerSessionPolicy**

String value indicating whether the game session accepts new players.

Type: `Aws::GameLift::Model::PlayerSessionCreationPolicy` enum. Valid values include:

- **ACCEPT_ALL** – Accept all new player sessions.
- **DENY_ALL** – Deny all new player sessions.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example sets the current game session's join policy to accept all players.

```cpp
Aws::GameLift::GenericOutcome outcome =
    Aws::GameLift::Server::UpdatePlayerSessionCreationPolicy(Aws::GameLift::Model::PlayerSessionCreationPolicy::ACCEPT_ALL);
```
Amazon GameLift Server API (C++) Reference: Data Types

This Amazon GameLift C++ Server API reference can help you prepare your multiplayer game for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).

This API is defined in GameLiftServerAPI.h, LogParameters.h, and ProcessParameters.h.

- Actions (p. 233)
- Data Types

DescribePlayerSessionsRequest

This data type is used to specify which player session(s) to retrieve. You can use it as follows:

- Provide a PlayerSessionId to request a specific player session.
- Provide a GameSessionId to request all player sessions in the specified game session.
- Provide a PlayerId to request all player sessions for the specified player.

For large collections of player sessions, use the pagination parameters to retrieve results in sequential blocks.

Contents

GameSessionId

Unique game session identifier. Use this parameter to request all player sessions for the specified game session. Game session ID format is as follows:
arn:aws:gamelift:<region>::gamesession/fleet-<fleet ID>/<ID string>. The value of <ID string> is either a custom ID string or (if one was specified when the game session was created) a generated string.

Type: String

Required: No

Limit

Maximum number of results to return. Use this parameter with NextToken to get results as a set of sequential pages. If a player session ID is specified, this parameter is ignored.

Type: Integer

Required: No

NextToken

Token indicating the start of the next sequential page of results. Use the token that is returned with a previous call to this action. To specify the start of the result set, do not specify a value. If a player session ID is specified, this parameter is ignored.

Type: String

Required: No

PlayerId

Unique identifier for a player. Player IDs are defined by the developer. See Generate Player IDs (p. 51).
**PlayerSessionId**

Unique identifier for a player session.

**Type:** String  
**Required:** No

**PlayerSessionStatusFilter**

Player session status to filter results on. Possible player session statuses include the following:

- **RESERVED** – The player session request has been received, but the player has not yet connected to the server process and/or has been validated.
- **ACTIVE** – The player has been validated by the server process and is currently connected.
- **COMPLETED** – The player connection has been dropped.
- **TIMEDOUT** – A player session request was received, but the player did not connect and/or was not validated within the time-out limit (60 seconds).

**Type:** String  
**Required:** No

**LogParameters**

This data type is used to identify which files generated during a game session that you want Amazon GameLift to upload and store once the game session ends. This information is communicated to the Amazon GameLift service in a `ProcessReady()` call.

**Contents**

**logPaths**

Directory paths to game server log files that you want Amazon GameLift to store for future access. These files are generated during each game session. File paths and names are defined in your game server and stored in the root game build directory. For example, if your game build stores game session logs in a path like `MyGame\sessionlogs\`, then the log path would be `c:\game\MyGame\sessionLogs` (on a Windows instance) or `/local/game/MyGame/sessionLogs` (on a Linux instance).

**Type:** `std::vector<std::string>`  
**Required:** No

**ProcessParameters**

This data type contains the set of parameters sent to the Amazon GameLift service in a `ProcessReady()` call.

**Contents**

**port**

Port number the server process listens on for new player connections. The value must fall into the port range configured for any fleet deploying this game server build. This port number is included in game session and player session objects, which game sessions use when connecting to a server process.
Type: Integer
Required: Yes

**logParameters**

Object with a list of directory paths to game session log files.

Type: `Aws::GameLift::Server::LogParameters` (p. 245)
Required: No

**onStartGameSession**

Name of callback function that the Amazon GameLift service calls to activate a new game session. Amazon GameLift calls this function in response to the client request `CreateGameSession`. The callback function passes a `GameSession` object (defined in the Amazon GameLift Service API Reference).

Type: `const std::function<void(Aws::GameLift::Model::GameSession)>` onStartGameSession

Required: Yes

**onProcessTerminate**

Name of callback function that the Amazon GameLift service calls to force the server process to shut down. After calling this function, Amazon GameLift waits five minutes for the server process to shut down and respond with a `ProcessEnding()` (p. 237) call. If no response is received, it shuts down the server process.

Type: `std::function<void()>` onProcessTerminate

Required: No

**onHealthCheck**

Name of callback function that the Amazon GameLift service calls to request a health status report from the server process. Amazon GameLift calls this function every 60 seconds. After calling this function Amazon GameLift waits 60 seconds for a response, and if none is received, records the server process as unhealthy.

Type: `std::function<bool()>` onHealthCheck

Required: No

**onUpdateGameSession**

Name of callback function that the Amazon GameLift service calls to provide an updated game session object. Amazon GameLift calls this function once a match backfill (p. 76) request has been processed. It passes a `GameSession` object, a status update (updateReason), and the match backfill ticket ID.

Type: `std::function<void(Aws::GameLift::Server::Model::UpdateGameSession)>` onUpdateGameSession

Required: No

**StartMatchBackfillRequest**

This data type is used to send a matchmaking backfill request. The information is communicated to the Amazon GameLift service in a `StartMatchBackfill()` (p. 241) call.
Contents

GameSessionArn

Unique game session identifier. The API action GetGameSessionId() (p. 235) returns the identifier in ARN format.

Type: String

Required: Yes

MatchmakingConfigurationArn

Unique identifier, in the form of an ARN, for the matchmaker to use for this request. To find the matchmaker that was used to create the original game session, look in the game session object, in the matchmaker data property. Learn more about matchmaker data in Work with Matchmaker Data (p. 75).

Type: String

Required: Yes

Players

A set of data representing all players who are currently in the game session. The matchmaker uses this information to search for new players who are good matches for the current players. See the Amazon GameLift API Reference Guide for a description of the Player object format. To find player attributes, IDs, and team assignments, look in the game session object, in the matchmaker data property. If latency is used by the matchmaker, gather updated latency for the current region and include it in each player's data.

Type: std::vector<Player>

Required: Yes

TicketId

Unique identifier for a matchmaking or match backfill request ticket. If no value is provided here, Amazon GameLift will generate one in the form of a UUID. Use this identifier to track the match backfill ticket status or cancel the request if needed.

Type: String

Required: No

StopMatchBackfillRequest

This data type is used to cancel a matchmaking backfill request. The information is communicated to the Amazon GameLift service in a StopMatchBackfill() (p. 242) call.

Contents

GameSessionArn

Unique game session identifier associated with the request being canceled.

Type: String

Required: Yes

MatchmakingConfigurationArn

Unique identifier of the matchmaker this request was sent to.
**TicketId**

Unique identifier of the backfill request ticket to be canceled.

Type: String

Required: Yes

### Amazon GameLift Server API (C#) Reference

This Amazon GameLift C# Server API reference can help you prepare your multiplayer game for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).

This API is defined in `GameLiftServerAPI.cs`, `LogParameters.cs`, and `ProcessParameters.cs`.

- Actions (p. 248)
- Data Types (p. 257)

### Amazon GameLift Server API (C#) Reference: Actions

This Amazon GameLift C# Server API reference can help you prepare your multiplayer game for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).

This API is defined in `GameLiftServerAPI.cs`, `LogParameters.cs`, and `ProcessParameters.cs`.

- Actions
- Data Types (p. 257)

**AcceptPlayerSession()**

Notifies the Amazon GameLift service that a player with the specified player session ID has connected to the server process and needs validation. Amazon GameLift verifies that the player session ID is valid—that is, that the player ID has reserved a player slot in the game session. Once validated, Amazon GameLift changes the status of the player slot from RESERVED to ACTIVE.

**Syntax**

```
GenericOutcome AcceptPlayerSession(String playerSessionId)
```

**Parameters**

**playerSessionId**

Unique ID issued by GameLift when a new player session is created. A player session ID is specified in a `PlayerSession` object, which is returned in response to a client call to the AWS SDK Amazon GameLift API actions `StartGameSessionPlacement`, `CreateGameSession`, `DescribeGameSessionPlacement`, or `DescribePlayerSessions`.

Type: String

Required: No
Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates a function for handling a connection request, including validating and rejecting invalid player session IDs.

```csharp
void ReceiveConnectingPlayerSessionID (Connection connection, String playerSessionId)
{
    var acceptPlayerSessionOutcome = GameLiftServerAPI.AcceptPlayerSession(playerSessionId);
    if(acceptPlayerSessionOutcome.Success)
    {
        connectionToSessionMap.emplace(connection, playerSessionId);
        connection.Accept();
    }
    else
    {
        connection.Reject(acceptPlayerSessionOutcome.Error.ErrorMessage);
    }
}
```

ActivateGameSession()

Notifies the Amazon GameLift service that the server process has activated a game session and is now ready to receive player connections. This action should be called as part of the `onStartGameSession()` callback function, after all game session initialization has been completed.

Syntax

```csharp
GenericOutcome ActivateGameSession()
```

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example shows `ActivateGameSession()` being called as part of the `onStartGameSession()` delegate function.

```csharp
void OnStartGameSession(GameSession gameSession)
{
    // game-specific tasks when starting a new game session, such as loading map
    // When ready to receive players
    var activateGameSessionOutcome = GameLiftServerAPI.ActivateGameSession();
}
```

DescribePlayerSessions()

Retrieves player session data, including settings, session metadata, and player data. Use this action to get information for a single player session, for all player sessions in a game session, or for all player sessions associated with a single player ID.
Syntax

DescribePlayerSessionsOutcome DescribePlayerSessions(DescribePlayerSessionsRequest describePlayerSessionsRequest)

Parameters

describePlayerSessionsRequest

A DescribePlayerSessionsRequest object describing which player sessions to retrieve.

Required: Yes

Return Value

If successful, returns a DescribePlayerSessionsOutcome object containing a set of player session objects that fit the request parameters. Player session objects have a structure identical to the AWS SDK Amazon GameLift API PlayerSession data type.

Example

This example illustrates a request for all player sessions actively connected to a specified game session. By omitting NextToken and setting the Limit value to 10, Amazon GameLift will return the first 10 player sessions records matching the request.

```
// Set request parameters
var describePlayerSessionsRequest = new
    {
        GameSessionId = GameLiftServerAPI.GetGameSessionId().Result,    //gets the ID for the
current game session
        Limit = 10,
        PlayerSessionStatusFilter = PlayerSessionStatusMapper.GetNameForPlayerSessionStatus(PlayerSessionStatus.ACTIVE)
    };
// Call DescribePlayerSessions
Aws::GameLift::DescribePlayerSessionsOutcome playerSessionsOutcome =
    Aws::GameLift::Server::Model::DescribePlayerSessions(describePlayerSessionRequest);
```

GetGameSessionId()

Retrieves the ID of the game session currently being hosted by the server process, if the server process is active.

Syntax

AwsStringOutcome GetGameSessionId()

Parameters

This action has no parameters.

Return Value

If successful, returns the game session ID as an AwsStringOutcome object. If not successful, returns an error message.
Example

```csharp
var getGameSessionIdOutcome = GameLiftServerAPI.GetGameSessionId();
```

**GetInstanceCertificate()**

Retrieves the file location of a pem-encoded TLS certificate that is associated with the fleet and its instances. This certificate is generated when a new fleet is created with the certificate configuration set to GENERATED. Use this certificate to establish a secure connection with a game client and to encrypt client/server communication.

**Syntax**

```csharp
GetInstanceCertificateOutcome GetInstanceCertificate();
```

**Parameters**

This action has no parameters.

**Return Value**

If successful, returns a `GetInstanceCertificateOutcome` object containing the location of the fleet's TLS certificate file, which is stored on the instance. If not successful, returns an error message.

Example

```csharp
var getInstanceCertificateOutcome = GameLiftServerAPI.GetInstanceCertificate();
```

**GetSdkVersion()**

Returns the current version number of the SDK built into the server process.

**Syntax**

```csharp
AwsStringOutcome GetSdkVersion()
```

**Parameters**

This action has no parameters.

**Return Value**

If successful, returns the current SDK version as an `AwsStringOutcome` object. The returned string includes the version number only (ex. "3.1.5"). If not successful, returns an error message.

Example

```csharp
var getSdkVersionOutcome = GameLiftServerAPI.GetSdkVersion();
```

**GetTerminationTime()**

Returns the time that a server process is scheduled to be shut down, if a termination time is available. A server process takes this action after receiving an `onProcessTerminate()` callback from the Amazon GameLift service. A server process may be shut down for several reasons: (1) process poor health, (2) when an instance is being terminated during a scale-down event, or (3) when an instance is being terminated due to a spot-instance interruption (p. 103).
If the process has received an `onProcessTerminate()` callback, the value returned is the estimated termination time in epoch seconds. If no termination time is available, the value returned is -1, which indicates that the server process may be terminated at any time. If the process has not received an `onProcessTerminate()` callback, the returned value will always be -1. Learn more about shutting down a server process (p. 45).

**Syntax**

```csharp
AwsLongOutcome GetTerminationTime()
```

**Parameters**

This action has no parameters.

**Return Value**

If successful, returns the current SDK version as an `AwsLongOutcome` object. The value is either the termination time in epoch seconds, or the value -1. If not successful, returns an error message.

**Example**

```csharp
var getTerminationTimeOutcome = GameLiftServerAPI.GetTerminationTime();
```

**InitSDK()**

Initializes the Amazon GameLift SDK. This method should be called on launch, before any other Amazon GameLift-related initialization occurs.

**Syntax**

```csharp
InitSDKOutcome InitSDK()
```

**Parameters**

This action has no parameters.

**Return Value**

If successful, returns an `InitSdkOutcome` object indicating that the server process is ready to call `ProcessReady()` (p. 253).

**Example**

```csharp
var initSDKOutcome = GameLiftServerAPI.InitSDK();
```

**ProcessEnding()**

Notifies the Amazon GameLift service that the server process is shutting down. This method should exit with an exit code of 0; a non-zero exit code results in an event message that the process did not exit cleanly.

**Syntax**

```csharp
GenericOutcome ProcessEnding()
```

**Parameters**

This action has no parameters.
Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

```csharp
var processEndingOutcome = GameLiftServerAPI.ProcessEnding();
```

ProcessReady()

Notifies the Amazon GameLift service that the server process is ready to host game sessions. This method should be called after successfully invoking `InitSDK()` (p. 252) and completing any setup tasks required before the server process can host a game session.

Syntax

```csharp
GenericOutcome ProcessReady(ProcessParameters processParameters)
```

Parameters

**processParameters**

A `ProcessParameters` (p. 259) object communicating the following information about the server process:

- Names of callback methods, implemented in the game server code, that the Amazon GameLift service invokes to communicate with the server process.
- Port number that the server process is listening on.
- Path to any game session-specific files that you want Amazon GameLift to capture and store.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates both the `ProcessReady()` (p. 253) call and delegate function implementations.

```csharp
// Set parameters and call ProcessReady
var processParams = new ProcessParameters(
    this.OnGameSession,
    this.OnProcessTerminate,
    this.OnHealthCheck,
    this.OnGameSessionUpdate,
    port,
    new LogParameters(new List<string>()          // Examples of log and error files written
        {                                          // by the game server
        "C:\game\logs",
        "C:\game\error"
    })
);

var processReadyOutcome = GameLiftServerAPI.ProcessReady(processParams);

// Implement callback functions
void OnGameSession(GameSession gameSession)
```
{ // game-specific tasks when starting a new game session, such as loading map // When ready to receive players
    var activateGameSessionOutcome = GameLiftServerAPI.ActivateGameSession();
}

void OnProcessTerminate()
{
    // game-specific tasks required to gracefully shut down a game session,
    // such as notifying players, preserving game state data, and other cleanup
    var ProcessEndingOutcome = GameLiftServerAPI.ProcessEnding();
    GameLiftServerAPI.Destroy();
}

bool OnHealthCheck()
{
    bool isHealthy;
    // complete health evaluation within 60 seconds and set health
    return isHealthy;
}

**RemovePlayerSession()**

Notifies the Amazon GameLift service that a player with the specified player session ID has disconnected from the server process. In response, Amazon GameLift changes the player slot to available, which allows it to be assigned to a new player.

**Syntax**

```csharp
GenericOutcome RemovePlayerSession(String playerSessionId)
```

**Parameters**

**playerSessionId**

Unique ID issued by GameLift when a new player session is created. A player session ID is specified in a PlayerSession object, which is returned in response to a client call to the AWS SDK Amazon GameLift API actions StartGameSessionPlacement, CreateGameSession, DescribeGameSessionPlacement, or DescribePlayerSessions.

Type: String
Required: No

**Return Value**

Returns a generic outcome consisting of success or failure with an error message.

**Example**

```csharp
Aws::GameLift::GenericOutcome disconnectOutcome =
    Aws::GameLift::Server::RemovePlayerSession(playerSessionId);
```

**StartMatchBackfill()**

Sends a request to find new players for open slots in a game session created with FlexMatch. See also the AWS SDK action StartMatchBackfill(). With this action, match backfill requests can be initiated by a game server process that is hosting the game session. Learn more about the FlexMatch backfill feature in Backfill Existing Games with FlexMatch (p. 76).

Version

254
This action is asynchronous. If new players are successfully matched, the Amazon GameLift service delivers updated matchmaker data using the callback function `OnUpdateGameSession()`. A server process can have only one active match backfill request at a time. To send a new request, first call `StopMatchBackfill()` (p. 256) to cancel the original request.

### Syntax

```
StartMatchBackfillOutcome StartMatchBackfill (StartMatchBackfillRequest startBackfillRequest);
```

### Parameters

**StartMatchBackfillRequest**

A `StartMatchBackfillRequest` (p. 260) object that communicates the following information:

- A ticket ID to assign to the backfill request. This information is optional; if no ID is provided, Amazon GameLift will autogenerate one.
- The matchmaker to send the request to. The full configuration ARN is required. This value can be acquired from the game session's matchmaker data.
- The ID of the game session that is being backfilled.
- Available matchmaking data for the game session's current players.

Required: Yes

### Return Value

Returns a `StartMatchBackfillOutcome` object with the match backfill ticket ID or failure with an error message.

### Example

```csharp
// Build a backfill request
var startBackfillRequest = new AWS.GameLift.Server.Model.StartMatchBackfillRequest()
{
    TicketId = "a ticket ID", //optional
    MatchmakingConfigurationArn = "the matchmaker configuration ARN",
    GameSessionId = GameLiftServerAPI.GetGameSessionId().Result, // gets ID for current game session
    //get player data for all currently connected players
    MatchmakerData = MatchmakerData.FromJson(gameSession.MatchmakerData); // gets matchmaker data for current players
    // get matchmakerData.Players
    // remove data for players who are no longer connected
    Players = ListOfPlayersRemainingInTheGame
};

// Send backfill request
var startBackfillOutcome = GameLiftServerAPI.StartMatchBackfill(startBackfillRequest);

// Implement callback function for backfill
void OnUpdateGameSession(GameSession myGameSession)
{
    // game-specific tasks to prepare for the newly matched players and update matchmaker data as needed
}
```
StopMatchBackfill()

Cancels an active match backfill request that was created with StartMatchBackfill() (p. 254). See also the AWS SDK action StopMatchmaking(). Learn more about the FlexMatch backfill feature in Backfill Existing Games with FlexMatch (p. 76).

Syntax

```
GenericOutcome StopMatchBackfill (StopMatchBackfillRequest stopBackfillRequest);
```

Parameters

**StopMatchBackfillRequest**

A StopMatchBackfillRequest (p. 261) object identifying the matchmaking ticket to cancel:

- ticket ID assigned to the backfill request being canceled
- matchmaker the backfill request was sent to
- game session associated with the backfill request

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

```
// Set backfill stop request parameters
var stopBackfillRequest = new AWS.GameLift.Server.Model.StopMatchBackfillRequest()
{
    TicketId = "a ticket ID", //optional, if not provided one is autogenerated
    MatchmakingConfigurationArn = "the matchmaker configuration ARN", //from the game
    GameSessionId = GameLiftServerAPI.GetGameSessionId().Result  //gets the ID for the
};
var stopBackfillOutcome = GameLiftServerAPI.StopMatchBackfillRequest(stopBackfillRequest);
```

TerminateGameSession()

Notifies the Amazon GameLift service that the server process has shut down the game session. (Currently, each server process hosts only one game session at a time, so there's no need to specify which session.) This action should be called at the end of the game session shutdown process. After calling this action, the server process can call ProcessReady() (p. 253) to signal its availability to host a new game session. Alternatively, it can call ProcessEnding() (p. 252) to shut down the server process and terminate the instance.

Syntax

```
GenericOutcome TerminateGameSession();
```

Parameters

This action has no parameters.
Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates a server process at the end of a game session.

```csharp
// game-specific tasks required to gracefully shut down a game session,
// such as notifying players, preserving game state data, and other cleanup
var terminateGameSessionOutcome = GameLiftServerAPI.TerminateGameSession();
var processReadyOutcome = GameLiftServerAPI.ProcessReady(processParams);
```

UpdatePlayerSessionCreationPolicy()

Updates the current game session's ability to accept new player sessions. A game session can be set to either accept or deny all new player sessions. (See also the UpdateGameSession() action in the Amazon GameLift Service API Reference).

Syntax

```csharp
GenericOutcome UpdatePlayerSessionCreationPolicy(PlayerSessionCreationPolicy playerSessionPolicy)
```

Parameters

**newPlayerSessionPolicy**

String value indicating whether the game session accepts new players.

Type: `PlayerSessionCreationPolicy` enum. Valid values include:

- **ACCEPT_ALL** – Accept all new player sessions.
- **DENY_ALL** – Deny all new player sessions.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example sets the current game session's join policy to accept all players.

```csharp
var updatePlayerSessionCreationPolicyOutcome = 
    GameLiftServerAPI.UpdatePlayerSessionCreationPolicy(PlayerSessionCreationPolicy.ACCEPT_ALL);
```

Amazon GameLift Server API (C#) Reference: Data Types

This Amazon GameLift C# Server API reference can help you prepare your multiplayer game for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).

This API is defined in `GameLiftServerAPI.cs`, `LogParameters.cs`, and `ProcessParameters.cs`.

- Actions (p. 248)
• Data Types

**LogParameters**

This data type is used to identify which files generated during a game session that you want Amazon GameLift to upload and store once the game session ends. This information is communicated to the Amazon GameLift service in a `ProcessReady()` (p. 253) call.

**Contents**

**logPaths**

List of directory paths to game server log files you want Amazon GameLift to store for future access. These files are generated by a server process during each game session; file paths and names are defined in your game server and stored in the root game build directory. For example, if your game build stores game session logs in a path like `\MyGame\sessionLogs\`, then the log path would be `c:\game\MyGame\sessionLogs` (on a Windows instance) or `/local/game/MyGame/sessionLogs` (on a Linux instance).

Type: `List<String>`

Required: No

**DescribePlayerSessionsRequest**

This data type is used to specify which player session(s) to retrieve. It can be used in several ways: (1) provide a `PlayerSessionId` to request a specific player session; (2) provide a `GameSessionId` to request all player sessions in the specified game session; or (3) provide a `PlayerId` to request all player sessions for the specified player. For large collections of player sessions, use the pagination parameters to retrieve results as sequential pages.

**Contents**

**GameSessionId**

Unique game session identifier. Use this parameter to request all player sessions for the specified game session. Game session ID format is as follows: `arn:aws:gamelift:<region>::gamesession/fleet-<fleet ID>/<ID string>`. The value of `<ID string>` is either a custom ID string (if one was specified when the game session was created) or a generated string.

Type: String

Required: No

**Limit**

Maximum number of results to return. Use this parameter with `NextToken` to get results as a set of sequential pages. If a player session ID is specified, this parameter is ignored.

Type: Integer

Required: No

**NextToken**

Token indicating the start of the next sequential page of results. Use the token that is returned with a previous call to this action. To specify the start of the result set, do not specify a value. If a player session ID is specified, this parameter is ignored.

Type: String
Required: No

**PlayerId**

Unique identifier for a player. Player IDs are defined by the developer. See [Generate Player IDs (p. 51)](#).

Type: String  
Required: No

**PlayerSessionId**

Unique identifier for a player session.

Type: String  
Required: No

**PlayerSessionStatusFilter**

Player session status to filter results on. Possible player session statuses include the following:

- **RESERVED** – The player session request has been received, but the player has not yet connected to the server process and/or been validated.
- **ACTIVE** – The player has been validated by the server process and is currently connected.
- **COMPLETED** – The player connection has been dropped.
- **TIMEDOUT** – A player session request was received, but the player did not connect and/or was not validated within the time-out limit (60 seconds).

Type: String  
Required: No

**ProcessParameters**

This data type contains the set of parameters sent to the Amazon GameLift service in a [ProcessReady() (p. 253)](#) call.

**Contents**

**port**

Port number the server process will listen on for new player connections. The value must fall into the port range configured for any fleet deploying this game server build. This port number is included in game session and player session objects, which game sessions use when connecting to a server process.

Type: Integer  
Required: Yes

**logParameters**

Object with a list of directory paths to game session log files.

Type: [Aws::GameLift::Server::LogParameters (p. 258)](#)  
Required: Yes

**onStartGameSession**

Name of callback function that the Amazon GameLift service calls to activate a new game session. Amazon GameLift calls this function in response to the client request [CreateGameSession](#). The
callback function takes a GameSession object (defined in the Amazon GameLift Service API Reference).

Type: void OnStartGameSessionDelegate(GameSession gameSession)
Required: Yes

onProcessTerminate

Name of callback function that the Amazon GameLift service calls to force the server process to shut down. After calling this function, Amazon GameLift waits five minutes for the server process to shut down and respond with a ProcessEnding() (p. 252) call before it shuts down the server process.

Type: void OnProcessTerminateDelegate()
Required: Yes

onHealthCheck

Name of callback function that the Amazon GameLift service calls to request a health status report from the server process. Amazon GameLift calls this function every 60 seconds. After calling this function Amazon GameLift waits 60 seconds for a response, and if none is received, records the server process as unhealthy.

Type: bool OnHealthCheckDelegate()
Required: Yes

onUpdateGameSession

Name of callback function that the Amazon GameLift service calls to provide an updated game session object. Amazon GameLift calls this function once a match backfill request has been processed. It passes a GameSession object, a status update (updateReason), and the match backfill ticket ID.

Type: void OnUpdateGameSessionDelegate ( UpdateGameSession updateGameSession )
Required: No

StartMatchBackfillRequest

This data type is used to send a matchmaking backfill request. The information is communicated to the Amazon GameLift service in a StartMatchBackfill() (p. 254) call.

Contents

GameSessionArn

Unique game session identifier. The API action GetGameSessionId() (p. 250) returns the identifier in ARN format.

Type: String
Required: Yes

MatchmakingConfigurationArn

Unique identifier, in the form of an ARN, for the matchmaker to use for this request. To find the matchmaker that was used to create the original game session, look in the game session object, in the matchmaker data property. Learn more about matchmaker data in Work with Matchmaker Data (p. 75).

Type: String
Players

A set of data representing all players who are currently in the game session. The matchmaker uses this information to search for new players who are good matches for the current players. See the Amazon GameLift API Reference Guide for a description of the Player object format. To find player attributes, IDs, and team assignments, look in the game session object, in the matchmaker data property. If latency is used by the matchmaker, gather updated latency for the current region and include it in each player’s data.

Type: Player[]

Required: Yes

TicketId

Unique identifier for a matchmaking or match backfill request ticket. If no value is provided here, Amazon GameLift will generate one in the form of a UUID. Use this identifier to track the match backfill ticket status or cancel the request if needed.

Type: String

Required: No

StopMatchBackfillRequest

This data type is used to cancel a matchmaking backfill request. The information is communicated to the Amazon GameLift service in a StopMatchBackfill() (p. 256) call.

Contents

GameSessionArn

Unique game session identifier associated with the request being canceled.

Type: String

Required: Yes

MatchmakingConfigurationArn

Unique identifier of the matchmaker this request was sent to.

Type: String

Required: Yes

TicketId

Unique identifier of the backfill request ticket to be canceled.

Type: String

Required: Yes

Amazon GameLift Server API Reference for Unreal Engine

This Amazon GameLift Server API reference can help you prepare your Unreal Engine game projects for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).
This API is defined in GameLiftServerSDK.h and GameLiftServerSDKModels.h.

To set up the Unreal Engine plugin and see code examples Add Amazon GameLift to an Unreal Engine Game Server Project (p. 36).

- Actions (p. 262)
- Data Types (p. 267)

Amazon GameLift Server API Reference for Unreal Engine: Actions

This Amazon GameLift Server API reference can help you prepare your Unreal Engine game projects for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).

This API is defined in GameLiftServerSDK.h and GameLiftServerSDKModels.h.

To set up the Unreal Engine plugin and see code examples Add Amazon GameLift to an Unreal Engine Game Server Project (p. 36).

- Actions
- Data Types (p. 267)

AcceptPlayerSession()

Notifies the Amazon GameLift service that a player with the specified player session ID has connected to the server process and needs validation. Amazon GameLift verifies that the player session ID is valid—that is, that the player ID has reserved a player slot in the game session. Once validated, Amazon GameLift changes the status of the player slot from RESERVED to ACTIVE.

Syntax

| FGameliftGenericOutcome AcceptPlayerSession(const FString& playerSessionId) |

Parameters

playerSessionId

Unique ID issued by the Amazon GameLift service in response to a call to the AWS SDK Amazon GameLift API action CreatePlayerSession. The game client references this ID when connecting to the server process.

Type: FString

Required: No

Return Value

Returns a generic outcome consisting of success or failure with an error message.

ActivateGameSession()

Notifies the Amazon GameLift service that the server process has activated a game session and is now ready to receive player connections. This action should be called as part of the onStartGameSession() callback function, after all game session initialization has been completed.
Syntax

FGameLiftGenericOutcome ActivateGameSession()

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

GetGameSessionId()

Retrieves the ID of the game session currently being hosted by the server process, if the server process is active.

Syntax

FGameLiftStringOutcome GetGameSessionId()

Parameters

This action has no parameters.

Return Value

If successful, returns the game session ID as an FGameLiftStringOutcome object. If not successful, returns an error message.

GetSdkVersion()

Returns the current version number of the SDK built into the server process.

Syntax

FGameLiftStringOutcome GetSdkVersion();

Parameters

This action has no parameters.

Return Value

If successful, returns the current SDK version as an FGameLiftStringOutcome object. The returned string includes the version number only (ex. "3.1.5"). If not successful, returns an error message.

Example

Aws::GameLift::AwsStringOutcome SdkVersionOutcome =
Aws::GameLift::Server::GetSdkVersion();

InitSDK()

Initializes the Amazon GameLift SDK. This method should be called on launch, before any other Amazon GameLift-related initialization occurs.
Syntax

FGameLiftGenericOutcome InitSDK()

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

ProcessEnding()

Notifies the Amazon GameLift service that the server process is shutting down. This method should be called after all other cleanup tasks, including shutting down all active game sessions. This method should exit with an exit code of 0; a non-zero exit code results in an event message that the process did not exit cleanly.

Syntax

FGameLiftGenericOutcome ProcessEnding()

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

ProcessReady()

Notifies the Amazon GameLift service that the server process is ready to host game sessions. This method should be called after successfully invoking InitSDK() (p. 263) and completing any setup tasks required before the server process can host a game session.

Syntax

FGameLiftGenericOutcome ProcessReady(FProcessParameters &processParameters)

Parameters

FProcessParameters

A FProcessParameters (p. 267) object communicating the following information about the server process:

• Names of callback methods, implemented in the game server code, that the Amazon GameLift service invokes to communicate with the server process.
• Port number that the server process is listening on.
• Path to any game session-specific files that you want Amazon GameLift to capture and store.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.
Example

See the sample code in Using the Unreal Engine Plugin (p. 38).

RemovePlayerSession()

Notifies the Amazon GameLift service that a player with the specified player session ID has disconnected from the server process. In response, Amazon GameLift changes the player slot to available, which allows it to be assigned to a new player.

Syntax

```cpp
FGameLiftGenericOutcome RemovePlayerSession(const FString& playerSessionId)
```

Parameters

playerSessionId

Unique ID issued by the Amazon GameLift service in response to a call to the AWS SDK Amazon GameLift API action CreatePlayerSession. The game client references this ID when connecting to the server process.

Type: FString

Required: No

Return Value

Returns a generic outcome consisting of success or failure with an error message.

StartMatchBackfill()

Sends a request to find new players for open slots in a game session created with FlexMatch. See also the AWS SDK action StartMatchBackfill(). With this action, match backfill requests can be initiated by a game server process that is hosting the game session. Learn more about the FlexMatch backfill feature in Backfill Existing Games with FlexMatch (p. 76).

A server process can have only one active match backfill request at a time. To send a new request, first call StopMatchBackfill() (p. 266) to cancel the original request.

Syntax

```cpp
FGameLiftStringOutcome StartMatchBackfill (FStartMatchBackfillRequest &startBackfillRequest);
```

Parameters

FStartMatchBackfillRequest

A FStartMatchBackfillRequest (p. 268) object that communicates the following information:

- A ticket ID to assign to the backfill request. This information is optional; if no ID is provided, Amazon GameLift will autogenerate one.
- The matchmaker to send the request to. The full configuration ARN is required. This value can be acquired from the game session's matchmaker data.
- The ID of the game session that is being backfilled.
- Available matchmaking data for the game session's current players.

Required: Yes
Return Value

If successful, returns the match backfill ticket as a FGameLiftStringOutcome object. If not successful, returns an error message. Ticket status can be tracked using the AWS SDK action DescribeMatchmaking().

StopMatchBackfill()

Cancels an active match backfill request that was created with StartMatchBackfill() (p. 265). See also the AWS SDK action StopMatchmaking(). Learn more about the FlexMatch backfill feature in Backfill Existing Games with FlexMatch (p. 76).

Syntax

FGameLiftGenericOutcome StopMatchBackfill (FStopMatchBackfillRequest &stopBackfillRequest);

Parameters

StopMatchBackfillRequest

A FStopMatchBackfillRequest (p. 269) object identifying the matchmaking ticket to cancel:
- ticket ID assigned to the backfill request being canceled
- matchmaker the backfill request was sent to
- game session associated with the backfill request

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

TerminateGameSession()

Notifies the Amazon GameLift service that the server process has shut down the game session. (Currently, each server process hosts only one game session at a time, so there's no need to specify which session.) This action should be called at the end of the game session shutdown process. After calling this action, the server process can call ProcessReady() (p. 264) to signal its availability to host a new game session. Alternatively, it can call ProcessEnding() (p. 264) to shut down the server process and terminate the instance.

Syntax

FGameLiftGenericOutcome TerminateGameSession();

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

UpdatePlayerSessionCreationPolicy()

Updates the current game session's ability to accept new player sessions. A game session can be set to either accept or deny all new player sessions. (See also the UpdateGameSession() action in the Amazon GameLift Service API Reference).
Syntax

```c
FGameLiftGenericOutcome UpdatePlayerSessionCreationPolicy(EPlayerSessionCreationPolicy policy)
```

**Parameters**

**Policy**

Value indicating whether the game session accepts new players.

Type: `EPlayerSessionCreationPolicy` enum. Valid values include:

- **ACCEPT_ALL** – Accept all new player sessions.
- **DENY_ALL** – Deny all new player sessions.

Required: Yes

**Return Value**

Returns a generic outcome consisting of success or failure with an error message.

---

**Amazon GameLift Server API Reference for Unreal Engine: Data Types**

This Amazon GameLift Server API reference can help you prepare your Unreal Engine game projects for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 42).

This API is defined in `GameLiftServerSDK.h` and `GameLiftServerSDKModels.h`.

To set up the Unreal Engine plugin and see code examples Add Amazon GameLift to an Unreal Engine Game Server Project (p. 36).

- **Actions** (p. 262)
- **Data Types**

**FProcessParameters**

This data type contains the set of parameters sent to the Amazon GameLift service in a `ProcessReady()` (p. 264) call.

**Contents**

**port**

Port number the server process will listen on for new player connections. The value must fall into the port range configured for any fleet deploying this game server build. This port number is included in game session and player session objects, which game sessions use when connecting to a server process.

Type: Integer

Required: Yes

**logParameters**

Object with a list of directory paths to game session log files.

Type: TArray<FString>
onStartGameSession

Name of callback function that the Amazon GameLift service calls to activate a new game session. Amazon GameLift calls this function in response to the client request CreateGameSession. The callback function takes a GameSession object (defined in the Amazon GameLift Service API Reference).

Type: FOnStartGameSession

Required: Yes

onProcessTerminate

Name of callback function that the Amazon GameLift service calls to force the server process to shut down. After calling this function, Amazon GameLift waits five minutes for the server process to shut down and respond with a ProcessEnding() (p. 264) call before it shuts down the server process.

Type: FSimpleDelegate

Required: No

onHealthCheck

Name of callback function that the Amazon GameLift service calls to request a health status report from the server process. Amazon GameLift calls this function every 60 seconds. After calling this function Amazon GameLift waits 60 seconds for a response, and if none is received, records the server process as unhealthy.

Type: FOnHealthCheck

Required: No

FStartMatchBackfillRequest

This data type is used to send a matchmaking backfill request. The information is communicated to the Amazon GameLift service in a StartMatchBackfill() (p. 265) call.

Contents

GameSessionArn

Unique game session identifier. The API action GetGameSessionId() (p. 263) returns the identifier in ARN format.

Type: FString

Required: Yes

MatchmakingConfigurationArn

Unique identifier, in the form of an ARN, for the matchmaker to use for this request. To find the matchmaker that was used to create the original game session, look in the game session object, in the matchmaker data property. Learn more about matchmaker data in Work with Matchmaker Data (p. 75).

Type: FString

Required: Yes

Players

A set of data representing all players who are currently in the game session. The matchmaker uses this information to search for new players who are good matches for the current players. See the
For a description of the Player object format. To find player attributes, IDs, and team assignments, look in the game session object, in the matchmaker data property. If latency is used by the matchmaker, gather updated latency for the current region and include it in each player's data.

Type: TArray<FPlayer>
Required: Yes

TicketId
Unique identifier for a matchmaking or match backfill request ticket. If no value is provided here, Amazon GameLift will generate one in the form of a UUID. Use this identifier to track the match backfill ticket status or cancel the request if needed.

Type: FString
Required: No

FStopMatchBackfillRequest
This data type is used to cancel a matchmaking backfill request. The information is communicated to the Amazon GameLift service in a StopMatchBackfill() (p. 266) call.

Contents

GameSessionArn
Unique game session identifier associated with the request being canceled.

Type: FString
Required: Yes

MatchmakingConfigurationArn
Unique identifier of the matchmaker this request was sent to.

Type: FString
Required: Yes

TicketId
Unique identifier of the backfill request ticket to be canceled.

Type: FString
Required: Yes

Amazon GameLift FlexMatch Reference

See reference documentation for Amazon GameLift FlexMatch.

Topics
- FlexMatch Rule Set Schema (p. 270)
- FlexMatch Rules Language (p. 276)
- FlexMatch Matchmaking Events (p. 280)
FlexMatch Rule Set Schema

This topic documents the standard schema for a small-match rule set and a large-match rule set. Use the rules language, detailed in

Use the following property expression syntax when writing rules for FlexMatch rule sets.

Learn more about creating FlexMatch rules:

• Build a FlexMatch Rule Set (p. 141)

• Create Matchmaking Rule Sets (p. 147)
• FlexMatch Rule Set Examples (p. 148)

Rule Types

FlexMatch supports the following rule types. Each rule time has a set of properties, which are described here.

Distance rule (distance)

Distance rules measure the difference between two number values, such as the distance between skill levels. For example, a distance rule might require that all players be within two levels of each other.

Distance rule properties

• measurements – Player attribute value to measure distance for; must be a number value.
• referenceValue – Number value to measure distance against for a prospective match.
• minDistance/maxDistance – Maximum or minimum distance value allowed for a successful match.
• partyAggregation – How to handle multiple-player (party) requests. Valid options are to use the minimum (min), maximum (max), or average (avg) values for a request's players. Default is avg.

Comparison rule (comparison)

Comparison rules compare a player attribute value against another value. There are two types of comparison rules. The first type compares an attribute value to a reference value. Specify the reference value and any valid comparison operation. For example, the rule might require that matched players have skill level 24 or greater. The second type compares an attribute value across all players in a team or match. This type omits the reference value and specifies either equals or not-equals (all or none of the players have the same attribute value). For example, the rule might require that all players choose the same game map.

Comparison rule properties

• measurements – Player attribute value to compare.
• referenceValue – Value to evaluate the measurement against for a prospective match.
• operation – How to evaluate the measurement. Valid operations include: <, <=, =, !=, >, >=.
• partyAggregation – How to sort multiple-player (party) requests. Valid options are to use the minimum (min), maximum (max), or average (avg) values for a request's players. Default is avg.
Collection rule (collection)

Collection rules evaluate collections of player attribute values. A collection might contain attribute values for multiple players, a player attribute that is a collection (a string list), or both. For example, a collection rule might look at the collection of characters chosen by a team's players and require that it contain at least one of a certain character.

Collection rule properties

- **measurements** – Collection of player attribute values to evaluate. Attribute values must be string lists.
- **referenceValue** – Value or collection of values to evaluate the measurements against for a prospective match.
- **operation** – How to evaluate a measurement collection. Valid operations include:
  - **intersection** measures the number of values that are common in all players' collections. See the MapOverlap rule in Example 4: Use Explicit Sorting to Find Best Matches (p. 153).
  - **contains** measures the number of player attribute collections that contain a certain reference value. See the OverallMedicLimit rule in Example 3: Set Team-Level Requirements and Latency Limits (p. 152).
  - **reference_intersection_count** measures the intersection between a player attribute collection and a reference value collection. Each player attribute collection (string list) in the measurements is evaluated against the reference collection separately. This operation can be used to evaluate across different player attributes. See the OpponentMatch rule in Example 5: Find Intersections Across Multiple Player Attributes (p. 155).
- **minCount/maxCount** – Maximum or minimum count value allowed for a successful match.
- **partyAggregation** – How to sort multiple-player (party) requests. Valid options are to use the union or intersection of values for a request's players. Default is union.

Latency rule (latency)

Latency rules evaluate player latency settings for acceptable matches. For example, a rule might require all matched players must have a regional latency under a maximum limit. Currently, this is the only rule type you can use in a rule set for large matches, and setting maxLatency is the only available option.

Latency rule properties

- **maxLatency** – Highest acceptable latency value for a region. For each player, ignore all regions that exceed this latency.
- **maxDistance** – Maximum difference between the latency of each player and the distance reference value.
- **distanceReference** – Use with maxDistance. Number value to measure distance against for a successful match. For latency, this value is an aggregate of latency values for multiple players. Valid options are minimum (min) or average (avg) player latency value. (See the Property Expressions section.)
- **partyAggregation** – How to sort multiple-player (party) match requests. Valid options are to use the minimum (min), maximum (max), or average (avg) values for a request's players. Default is avg.

Distance sort rule (distanceSort)

Distance sort is an explicit sort option that directs the matchmaker to pre-sort matchmaking requests based on a player attribute. A distance sort rule evaluates matchmaking requests based on the distance from the anchor request.

Distance sort rule properties
• **sortDirection** – Direction to sort matchmaking requests. Valid options are ascending or descending.

• **sortAttribute** – Player attribute to sort players by.

• **mapKey** – How to evaluate the player attribute if it’s a map. Valid options include:
  • **minValue**: For the anchor player, find the key with the lowest value.
  • **maxValue**: For the anchor player, find the key with the highest value.

• **partyAggregation** – How to sort multiple-player (party) requests. Valid options are to use the minimum (min), maximum (max), or average (avg) values for a request’s players. Default is avg.

**Absolute sort rule (absoluteSort)**

Absolute sort is an explicit sort option that directs the matchmaker to pre-sort matchmaking requests based on a player attribute. An absolute sort evaluates matchmaking requests based on whether its player attribute matches that of the anchor request.

**Absolute sort rule properties**

• **sortDirection** – Direction to sort matchmaking requests. Valid options are ascending or descending.

• **sortAttribute** – Player attribute to sort players by.

• **mapKey** – How to evaluate the player attribute if it’s a map. Valid options include:
  • **minValue**: For the anchor player, find the key with the lowest value.
  • **maxValue**: For the anchor player, find the key with the highest value.

• **partyAggregation** – How to sort multiple-player (party) requests. Valid options are to use the minimum (min), maximum (max), or average (avg) values for a request’s players. Default is avg.

**Property Expressions**

Property expressions are used in a rule set to refer to certain matchmaking-related properties. Properties can include the player attribute values from matchmaking requests. For example, a rule might use a property expression to identify which player attribute to evaluate.

They generally take one of two forms:

• Individual player data

• Calculated team data, which takes the form of collections of individual player data. A valid property expression identifies a specific value for a single player, team, or match. The following partial expressions illustrate how to identify teams and players:

<table>
<thead>
<tr>
<th>To identify a specific team in a match:</th>
<th>teams[red]</th>
<th>The Red team</th>
<th>Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>To identify all teams in a match:</td>
<td>teams[*]</td>
<td>All teams</td>
<td>List&lt;Team&gt;</td>
</tr>
<tr>
<td>To identify players in a specific team:</td>
<td>team[red].players</td>
<td>Players in the Red team</td>
<td>List&lt;Player&gt;</td>
</tr>
<tr>
<td>To identify players in a match:</td>
<td>team[*].players</td>
<td>Players in the match, grouped by team</td>
<td>List&lt;List&lt;Player&gt;&gt;</td>
</tr>
</tbody>
</table>
The following table illustrates some valid property expressions that build on the previous examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
<th>Resulting Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>teams[red].players[playerid]</td>
<td>The player IDs of all players on the red team</td>
<td>List&lt;string&gt;</td>
</tr>
<tr>
<td>teams[red].players.attributes[skill]</td>
<td>&quot;skill&quot; attributes of all players on the red team</td>
<td>List&lt;number&gt;</td>
</tr>
<tr>
<td>teams[*].players.attributes[skill]</td>
<td>The &quot;skill&quot; attributes of all players in the match, grouped by team</td>
<td>List&lt;List&lt;number&gt;&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
<th>Resulting Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>List&lt;number&gt;</td>
<td>Get the minimum of all numbers in the list.</td>
</tr>
<tr>
<td>max</td>
<td>List&lt;number&gt;</td>
<td>Get the maximum of all numbers in the list.</td>
</tr>
<tr>
<td>avg</td>
<td>List&lt;number&gt;</td>
<td>Get the average of all numbers in the list.</td>
</tr>
<tr>
<td>median</td>
<td>List&lt;number&gt;</td>
<td>Get the median of all numbers in the list.</td>
</tr>
<tr>
<td>sum</td>
<td>List&lt;number&gt;</td>
<td>Get the sum of all numbers in the list.</td>
</tr>
<tr>
<td>count</td>
<td>List&lt;??&gt;</td>
<td>Get the number of elements in the list.</td>
</tr>
<tr>
<td>stddev</td>
<td>List&lt;number&gt;</td>
<td>Get the standard deviation of all numbers in the list.</td>
</tr>
<tr>
<td>flatten</td>
<td>List&lt;List&lt;??&gt;&gt;</td>
<td>Turn a collection of nested lists into a single list containing all elements.</td>
</tr>
<tr>
<td>set_intersection</td>
<td>List&lt;List&lt;string&gt;&gt;</td>
<td>Get a list of strings that are found in all string lists in a collection.</td>
</tr>
<tr>
<td>All above</td>
<td>List&lt;List&lt;??&gt;&gt;</td>
<td>All operations on a nested list operate on each sublist</td>
</tr>
</tbody>
</table>
Aggregation | Input | Meaning | Output
--- | --- | --- | ---
individually to | produce a list of results. | 

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Input</th>
<th>Meaning</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>flatten(teams[*].players.attributes[skill])</td>
<td>The &quot;skill&quot; attributes of all players in the match (not grouped)</td>
<td>List&lt;number&gt;</td>
<td></td>
</tr>
<tr>
<td>avg(teams[red].players.attributes[skill])</td>
<td>The average skill of the red team players</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>avg(teams[*].players.attributes[skill])</td>
<td>The average skill of each team in the match</td>
<td>List&lt;number&gt;</td>
<td></td>
</tr>
<tr>
<td>avg(flatten(teams[*].players.attributes[skill]))</td>
<td>The average skill level of all players in the match. This expression gets a flattened list of player skills and then averages them.</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>count(teams[red].players)</td>
<td>The number of players on the red team</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>count (teams[*].players)</td>
<td>The number of players on each team in the match</td>
<td>List&lt;number&gt;</td>
<td></td>
</tr>
<tr>
<td>max(avg(teams[*].players.attributes[skill]))</td>
<td>The highest team skill level in the match</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

(p. 276) to develop your custom values.

**Rule Set Schema for Small Matches**

Use this schema when creating a rule set to build matches of 40 players maximum.

```json
{
    "name": <descriptive label, string>,
    "ruleLanguageVersion": <must be "1.0">,
    "playerAttributes": [{
        "name": <unique name for player attribute to be used by matchmaker, string>,
        "type": <attribute data type, allowed values are "string", "number",
                "string_list", "string_number_map">,
        "default": <value to use when no player-specific value is provided>
    }]
}
```
Rule Set Schema for Large Matches

Use this schema when creating a rule set to build matches of greater than 40 players. If the `maxPlayers` values for all teams defined in the rule set exceeds 40, then GameLift processes all requests that use this rule sets under large-match guidelines.

```json
{
    "name": <descriptive label, string>,
    "ruleLanguageVersion": "1.0",
    "playerAttributes": [{
        "name": <unique name for player attribute to be used by matchmaker, string>,
        "type": <attribute data type, allowed values are "string", "number", "string_list", "string_number_map">,
        "default": <value to use when no player-specific value is provided>
    }],
    "teams": [{
        "name": <unique label, string>,
        "maxPlayers": <max players allowed in team>,
        "minPlayers": <min players required in team>,
        "quantity": <number of teams to create with this team definition>
    }],
    "algorithm": {
        "balancedAttribute": <name of player attribute, data type "number", to use when grouping players >,
        "strategy": "balanced",
        "batchingPreference": <choose between "largestPopulation" (default) or "fastestRegion">
    },
    "rules": [{
        "name": <unique label, string>,
        "description": <descriptive label, string>,
        "type": "latency",
        "maxLatency": <property expression, must set value for "maxLatency">
    }],
    "expansions": [{
        "target": <rule/team and property to adjust value for, example: "rules[<minSkill>].referenceValue">,
        "steps": [{
            "waitTimeSeconds": <length of 1st wait period before relaxing rule>,
            "value": <new value>
        }, {
            "waitTimeSeconds": <length of 2nd wait period before further relaxing rule>,
            "value": <new value>
        }]
    }]
}
```
FlexMatch Rules Language

Use the following property expression syntax when writing rules for FlexMatch rule sets.

Learn more about creating FlexMatch rules:

- Build a FlexMatch Rule Set (p. 141)
- Create Matchmaking Rule Sets (p. 147)
- FlexMatch Rule Set Examples (p. 148)

Rule Types

FlexMatch supports the following rule types. Each rule time has a set of properties, which are described here.

Distance rule (distance)

Distance rules measure the difference between two number values, such as the distance between skill levels. For example, a distance rule might require that all players be within two levels of each other.

Distance rule properties

- measurements – Player attribute value to measure distance for; must be a number value.
- referenceValue – Number value to measure distance against for a prospective match.
- minDistance/maxDistance – Maximum or minimum distance value allowed for a successful match.
- partyAggregation – How to handle multiple-player (party) requests. Valid options are to use the minimum (min), maximum (max), or average (avg) values for a request's players. Default is avg.

Comparison rule (comparison)

Comparison rules compare a player attribute value against another value. There are two types of comparison rules. The first type compares an attribute value to a reference value. Specify the reference value and any valid comparison operation. For example, the rule might require that matched players have skill level 24 or greater. The second type compares an attribute value across all players in a team or match. This type omits the reference value and specifies either equals or not-equals (all or none of the players have the same attribute value). For example, the rule might require that all players choose the same game map.

Comparison rule properties

- measurements – Player attribute value to compare.
- referenceValue – Value to evaluate the measurement against for a prospective match.
- operation – How to evaluate the measurement. Valid operations include: <, <=, =, !=, >, >=.
• **partyAggregation** – How to sort multiple-player (party) requests. Valid options are to use the
  minimum (min), maximum (max), or average (avg) values for a request's players. Default is avg.

**Collection rule (collection)**

Collection rules evaluate collections of player attribute values. A collection might contain attribute
values for multiple players, a player attribute that is a collection (a string list), or both. For example,
a collection rule might look at the collection of characters chosen by a team's players and require
that it contain at least one of a certain character.

**Collection rule properties**

• **measurements** – Collection of player attribute values to evaluate. Attribute values must be string
  lists.

• **referenceValue** – Value or collection of values to evaluate the measurements against for a
  prospective match.

• **operation** – How to evaluate a measurement collection. Valid operations include:
  • intersection measures the number of values that are common in all players' collections. See
    the MapOverlap rule in Example 4: Use Explicit Sorting to Find Best Matches (p. 153).
  • contains measures the number of player attribute collections that contain a certain reference
    value. See the OverallMedicLimit rule in Example 3: Set Team-Level Requirements and Latency
    Limits (p. 152).
  • reference_intersection_count measures the intersection between a player attribute
    collection and a reference value collection. Each player attribute collection (string list) in the
    measurements is evaluated against the reference collection separately. This operation can be
    used to evaluate across different player attributes. See the OpponentMatch rule in Example 5:
    Find Intersections Across Multiple Player Attributes (p. 155).

• **minCount/maxCount** – Maximum or minimum count value allowed for a successful match.

• **partyAggregation** – How to sort multiple-player (party) requests. Valid options are to use the
  union or intersection of values for a request's players. Default is union.

**Latency rule (latency)**

Latency rules evaluate player latency settings for acceptable matches. For example, a rule might
require all matched players must have a regional latency under a maximum limit. Currently, this is
the only rule type you can use in a rule set for large matches, and setting maxLatency is the only
available option.

**Latency rule properties**

• **maxLatency** – Highest acceptable latency value for a region. For each player, ignore all regions
  that exceed this latency.

• **maxDistance** – Maximum difference between the latency of each player and the distance
  reference value.

• **distanceReference** – Use with maxDistance. Number value to measure distance against
  for a successful match. For latency, this value is an aggregate of latency values for multiple
  players. Valid options are minimum (min) or average (avg) player latency value. (See the Property
  Expressions section.)

• **partyAggregation** – How to sort multiple-player (party) match requests. Valid options are to
  use the minimum (min), maximum (max), or average (avg) values for a request's players. Default is
  avg.

**Distance sort rule (distanceSort)**

Distance sort is an explicit sort option that directs the matchmaker to pre-sort matchmaking
requests based on a player attribute. A distance sort rule evaluates matchmaking requests based on
the distance from the anchor request.
Distance sort rule properties

- **sortDirection** – Direction to sort matchmaking requests. Valid options are ascending or descending.
- **sortAttribute** – Player attribute to sort players by.
- **mapKey** – How to evaluate the player attribute if it's a map. Valid options include:
  - **minValue**: For the anchor player, find the key with the lowest value.
  - **maxValue**: For the anchor player, find the key with the highest value.
- **partyAggregation** – How to sort multiple-player (party) requests. Valid options are to use the minimum (min), maximum (max), or average (avg) values for a request's players. Default is avg.

Absolute sort rule (**absoluteSort**) Property Expressions

Property Expressions are used in a rule set to refer to certain matchmaking-related properties. Properties can include the player attribute values from matchmaking requests. For example, a rule might use a property expression to identify which player attribute to evaluate.

They generally take one of two forms:

- Individual player data
- Calculated team data, which takes the form of collections of individual player data.

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

<table>
<thead>
<tr>
<th>To identify a specific team in a match:</th>
<th>teams[red]</th>
<th>The Red team</th>
<th>Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>To identify all teams in a match:</td>
<td>teams[*]</td>
<td>All teams</td>
<td>List&lt;Team&gt;</td>
</tr>
<tr>
<td>To identify players in a specific team:</td>
<td>team[red].players</td>
<td>Players in the Red team</td>
<td>List&lt;Player&gt;</td>
</tr>
<tr>
<td>To identify players in a match:</td>
<td>team[*].players</td>
<td>Players in the match, grouped by team</td>
<td>List&lt;List&lt;Player&gt;&gt;</td>
</tr>
</tbody>
</table>
The following table illustrates some valid property expressions that build on the previous examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
<th>Resulting Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>teams[red].players[playerid]</td>
<td>The player IDs of all players on the red team</td>
<td>List&lt;string&gt;</td>
</tr>
<tr>
<td>teams[red].players.attributes[skill]</td>
<td>The &quot;skill&quot; attributes of all players on the red team</td>
<td>List&lt;number&gt;</td>
</tr>
<tr>
<td>teams[*].players.attributes[skill]</td>
<td>The &quot;skill&quot; attributes of all players in the match, grouped by team</td>
<td>List&lt;List&lt;number&gt;&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Aggregation</th>
<th>Input</th>
<th>Meaning</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>List&lt;number&gt;</td>
<td>Get the minimum of all numbers in the list.</td>
<td>number</td>
</tr>
<tr>
<td>max</td>
<td>List&lt;number&gt;</td>
<td>Get the maximum of all numbers in the list.</td>
<td>number</td>
</tr>
<tr>
<td>avg</td>
<td>List&lt;number&gt;</td>
<td>Get the average of all numbers in the list.</td>
<td>number</td>
</tr>
<tr>
<td>median</td>
<td>List&lt;number&gt;</td>
<td>Get the median of all numbers in the list.</td>
<td>number</td>
</tr>
<tr>
<td>sum</td>
<td>List&lt;number&gt;</td>
<td>Get the sum of all numbers in the list.</td>
<td>number</td>
</tr>
<tr>
<td>count</td>
<td>List&lt;?&gt;</td>
<td>Get the number of elements in the list.</td>
<td>number</td>
</tr>
<tr>
<td>stddev</td>
<td>List&lt;number&gt;</td>
<td>Get the standard deviation of all numbers in the list.</td>
<td>number</td>
</tr>
<tr>
<td>flatten</td>
<td>List&lt;List&lt;?&gt;&gt;</td>
<td>Turn a collection of nested lists into a single list containing all elements.</td>
<td>List&lt;?&gt;&gt;</td>
</tr>
<tr>
<td>set_intersection</td>
<td>List&lt;List&lt;string&gt;&gt;</td>
<td>Get a list of strings that are found in all string lists in a collection.</td>
<td>List&lt;string&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
<th>Resulting Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>flatten(teams[*].players.attributes[skill])</td>
<td>The “skill” attributes of all players in the match (not grouped)</td>
<td>List&lt;number&gt;</td>
</tr>
<tr>
<td>avg(teams[red].players.attributes[skill])</td>
<td>The average skill of the red team players</td>
<td>number</td>
</tr>
<tr>
<td>avg(teams[*].players.attributes[skill])</td>
<td>The average skill of each team in the match</td>
<td>List&lt;number&gt;</td>
</tr>
<tr>
<td>avg(flatten(teams[*].players.attributes[skill]))</td>
<td>The average skill level of all players in the match. This expression gets a flattened list of player skills and then averages them.</td>
<td>number</td>
</tr>
<tr>
<td>count(teams[red].players)</td>
<td>The number of players on the red team</td>
<td>number</td>
</tr>
<tr>
<td>count (teams[*].players)</td>
<td>The number of players on each team in the match</td>
<td>List&lt;number&gt;</td>
</tr>
<tr>
<td>max(avg(teams[*].players.attributes[skill]))</td>
<td>The highest team skill level in the match</td>
<td>number</td>
</tr>
</tbody>
</table>

### FlexMatch Matchmaking Events

Amazon GameLift emits events that are related to the processing of matchmaking tickets. All the events that are listed here can be published to an Amazon SNS topic. These events are also emitted to Amazon CloudWatch Events. For more details working with matchmaking events, see Set up FlexMatch Event Notification (p. 163). For more information on matchmaking ticket statuses, see MatchmakingTicket in the Amazon GameLift Service 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**

```json
{
    "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"
    ]
}
```
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": [
```
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**

```json
{
   "version": "0",
   "id": "b3f76d66-c8e5-416a-aa4c-aa1278153edc",
   "detail-type": "GameLift Matchmaking Event",
   "source": "aws.gamelift",
   "playerId": "player-1",
   "team": "red",
   "playerId": "player-2",
   "team": "blue"
}
```
AcceptMatchCompleted

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

Resource: ConfigurationArn

Detail: type, tickets, acceptance, matchId, gameSessionInfo

Example

```json
{
    "version": "0",
    "id": "b1990d3d-f737-4d6c-b150-af5ace8c35d3",
    "detail-type": "GameLift Matchmaking Event",
    "source": "aws.gamelift",
    "account": "123456789012",
    "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"
    }
}
```
Matchmaking has successfully completed and a game session has been created.

**Resource:** ConfigurationArn  
**Detail:** type, tickets, matchId, gameSessionInfo

**Example**

```json
{
  "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",
```
MatchmakingTimedOut

Matchmaking ticket has failed by timing out.

**Resource:** ConfigurationArn

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

**Example**

```json
{
  "version": "0",
  "id": "fe528a7d-46ad-4bdc-b094b5f6b56",
```
"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"
            }
        ]
    }
},

Matchmaking Cancelled

Matchmaking ticket has been canceled.

Resource: ConfigurationArn

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

```json
{
    "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**

```json
{
  "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"
    }
  }
}
```
# Document History for Amazon GameLift

The following table describes important changes to the Amazon GameLift documentation. For details on releases of new and updated features, see the Amazon GameLift Release Notes.

<table>
<thead>
<tr>
<th>Date</th>
<th>Change</th>
<th>Documentation updates</th>
<th>API Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 15, 2019</td>
<td>Updates</td>
<td><strong>Developer Guide:</strong></td>
<td>AWS SDK: 2019-11-15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Expanded CloudFormation template support for GameLift – Existing CloudFormation templates for Build and Fleet have been updated and there are new templates Script, Queue, MatchmakingConfiguration, and MatchmakingRuleSet.</td>
<td>Server SDK 3.4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Developer Guide:</strong></td>
<td>Realtime Client SDK 1.1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New topic:</td>
<td>GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Create Resources Using AWS CloudFormation (p. 84)</td>
<td></td>
</tr>
<tr>
<td>October 24, 2019</td>
<td>Updates</td>
<td>**Developer Guide and API reference docs updated to reflect the additional options. Updates were released in the Global Regions on October 24, 2019, and released in the China (Beijing) Region, operated by Sinnet, on November 15, 2019.</td>
<td>AWS SDK: 2019-10-24</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Developer Guide:</strong></td>
<td>Server SDK 3.4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Support for Amazon Linux 2 and Series 5 instance types – Host games using the new C5/M5/R5 instance types. Game servers can now run on Amazon Linux 2. All Realtime Servers now run on Amazon Linux 2.</td>
<td>Realtime Client SDK 1.1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Developer Guide:</strong></td>
<td>GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td>September 3, 2019</td>
<td>New Features:</td>
<td><strong>Developer Guide:</strong></td>
<td>AWS SDK: 2019-09-03</td>
</tr>
<tr>
<td></td>
<td>• TLS certificate generation – GameLift can now generate fleet-level TLS certificates to support server authentication and data packet encryption for custom and Realtime servers.</td>
<td>• The following topics are updated with information about TLS certificates:</td>
<td>Server SDK 3.4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hosting Game Servers (p. 4)</td>
<td>Realtime Client SDK 1.1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• How Realtime Clients and Servers Interact (p. 10)</td>
<td>GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Retrieve a TLS Certificate (p. 44)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Integrating a Game Client for Realtime Servers (p. 59)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>July 9, 2019</td>
<td><strong>New Features:</strong></td>
<td><em>FlexMatch support for large matches</em> – Matchmaking expanded to support 200 players per match with new, faster algorithm. Match backfill now has an automatic mode.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Developer Guide:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FlexMatch topics updated or expanded to include information on working with large matches:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• How Amazon GameLift FlexMatch Works (p. 13)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Design a FlexMatch Rule Set (p. 142)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FlexMatch Rule Set Schema (p. 270)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FlexMatch Rule Set Examples (p. 148)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Backfill Existing Games with FlexMatch (p. 76)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Design a FlexMatch Matchmaker (p. 138)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Service API Reference for the AWS SDK:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Matchmaking configuration API calls updated to support automatic backfill:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CreateMatchmakingConfiguration, UpdateMatchmakingConfiguration, StopMatchmaking</td>
<td></td>
</tr>
</tbody>
</table>

**SDK & API References:**
- GameLift Service API, new CreateFleet() parameter CertificateConfiguration
- GameLift Server SDK, new action GetInstanceCertificate() C++ (p. 235) and C# (p. 251)
- Realtime Client SDK, new parameter the section called “Data Types” (p. 223)
<table>
<thead>
<tr>
<th>Date</th>
<th>Change</th>
<th>Documentation updates</th>
<th>API Versions</th>
</tr>
</thead>
</table>
| April 25, 2019 | **New Features:**  
                  • Realtime Servers—Use Realtime Servers with basic game server architecture built in to deploy your games. Add custom game server logic as needed.  
                  | **Developer Guide:**  
                  • New topics on Realtime Servers:  
                    • How Realtime Servers Work (p. 8)  
                    • Integrating Games with Amazon GameLift Realtime Servers (p. 59)  
                    • Realtime Servers Client API (C#) Reference (p. 217)  
                  | **Service API Reference for the AWS SDK:**  
                  • New API calls to manage Realtime Servers scripts: CreateScript, DescribeScript, UpdateScript, DeleteScript, ListScripts  
                  • Fleet API calls updated to enable Realtime Servers deployment with scripts.  
                  | AWS SDK: 2019-04-25  
                  Server SDK 3.3.0  
                  Realtime Client SDK 1.0.0  
                  GameLiftLocal 1.0.0 |
| March 7, 2019  | **New Features:**  
                  • Secure access to AWS resources—Allow applications running on an Amazon GameLift instance to communicate with AWS resources you own.  
                  | **Developer Guide:**  
                  • New topic Access AWS Resources From Your Fleets (p. 45) on secure access options.  
                  | **Service API Reference for the AWS SDK:**  
                  • CreateFleet and FleetAttributes – InstanceRoleArn added.  
                  | AWS SDK: 2019-03-07  
                  Server SDK 3.3.0  
                  GameLiftLocal 1.0.0 |
| February 7, 2019 | **New Features:**  
                  • Delete matchmaking rule sets  
                  | **Developer Guide:**  
                  • Create Matchmaking Rule Sets (p. 147) updated for matchmaking rule set deletion.  
                  | **Service API Reference for the AWS SDK:**  
                  • DeleteMatchmakingRuleSet (new)  
                  | AWS SDK: 2019-02-07  
                  Server SDK 3.3.0  
                  GameLiftLocal 1.0.0 |
<table>
<thead>
<tr>
<th>Date</th>
<th>Change</th>
<th>Documentation updates</th>
<th>API Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 14, 2018</td>
<td>Updates:</td>
<td>Developer Guide:</td>
<td>AWS SDK: 2018-12-14</td>
</tr>
<tr>
<td></td>
<td>• New Server SDK version</td>
<td>• Updated docs on building and using the Server SDK with Unity (p. 40) and Unreal (p. 36) games.</td>
<td>Server SDK 3.3.0 GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td></td>
<td>– Updated Server SDK</td>
<td>• SDK Compatibility (p. 25) updated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is now compatible with</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>latest Unity &amp; Unreal</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>engines and can be used</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with Visual Studio 15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2017.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FlexMatch Backfill</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>supported for Unreal</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>engine games.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Amazon GameLift in China</td>
<td>• New topic Using Amazon GameLift in AWS Regions (p. 27) provides links to information on using AWS in China.</td>
<td>Server SDK 3.2.1 GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td></td>
<td>– The Amazon GameLift</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>service is now available</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in the AWS China (Beijing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Region operated by Sinnet.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Queue metrics for FleetIQ</td>
<td>• Five new queue metrics added to Amazon GameLift Metrics for Queues (p. 183).</td>
<td>Server SDK 3.2.1 GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td></td>
<td>– Use FleetIQ metrics to</td>
<td>• New topic Evaluate Queue Metrics (p. 132) to optimize queue performance.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>track queue performance.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------</td>
<td>------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>May 10, 2018</td>
<td><strong>New Features:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Auto-scaling with Target Tracking – Use this new scaling method to match fleet capacity to player demand.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Enable/disable auto-scaling – Switch between auto-scaling and manual scaling without having to delete scaling policies.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Developer Guide:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• New section Scaling Amazon GameLift Fleet Capacity (p. 119) encompasses all scaling-related topics, including auto-scaling.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Updated section on Scaling Fleet Capacity (p. 6) in the &quot;How Amazon GameLift Works&quot; topic.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Service API Reference for the AWS SDK:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• PutScalingPolicy (updated with new examples)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• StopFleetActions and StartFleetActions (new)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>February 15, 2018</td>
<td><strong>New Features:</strong></td>
<td>• Spot Instances and FleetIQ – Use the new FleetIQ feature with spot fleets to significantly lower hosting costs.</td>
<td><strong>AWS SDK: 2018-02-15</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Server SDK 3.2.1</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>GameLiftLocal 1.0.0</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Developer Guide:</strong></td>
<td>• Spot Fleet Integration Guide (p. 103) – New step-by-step guide for spot fleet usage.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Design a Game Session Queue (p. 128) – New design tips for creating and using queues, including with spot fleets.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated topics:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Choose Computing Resources (p. 100)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deploy a GameLift Fleet for a Custom Game Build (p. 104)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shut Down a Server Process (p. 45)</td>
<td></td>
</tr>
</tbody>
</table>
|             | **Service API Reference for the AWS SDK:** | • New FleetType parameter added to enable spot fleets:  
  • CreateFleet  
  • FleetAttributes  
  • Event |                               |
<p>|             | <strong>Server SDK:</strong>      | • GetTerminationTime (C++) (p. 236) (new)                                               |                               |
|             |                      | GetTerminationTime (C#) (p. 251) (new)                                                 |                               |</p>
<table>
<thead>
<tr>
<th>Date</th>
<th>Change</th>
<th>Documentation updates</th>
<th>API Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>February 8, 2018</td>
<td><strong>New Features:</strong></td>
<td></td>
<td>AWS SDK: 2018-02-08</td>
</tr>
<tr>
<td></td>
<td>• FlexMatch Backfill – Use FlexMatch to inject new players into matched game sessions that are in progress.</td>
<td>Developer Guide:</td>
<td>Server SDK 3.2.0</td>
</tr>
<tr>
<td></td>
<td>• Game session search – use custom game properties</td>
<td>• Matchmaking Process (p. 14) – Updated FlexMatch feature overview.</td>
<td>GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Backfill Existing Games with FlexMatch (p. 76) – New how-to guide.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Get Game Sessions (p. 48) – Updated for custom game properties.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Service API Reference for the AWS SDK:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• StartMatchBackfill (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SearchGameSessions (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• GameSession (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Server SDK:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• StartMatchBackfill (C++) (p. 241) (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• StartMatchBackfill (C#) (p. 254) (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• StopMatchBackfill (C++) (p. 242) (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• StopMatchBackfill (C#) (p. 256) (new)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>---------------------------</td>
</tr>
</tbody>
</table>
| September 1, 2017 | **New Features:**  
VPC Peering for Amazon GameLift fleets – You can now use VPC peering to set up direct, private connections between your Amazon GameLift game servers and other resources on AWS. | **Developer Guide:**  
- Networking With AWS Resources (p. 8) – Get more information on VPC peering and how it works with Amazon GameLift.  
- VPC Peering for Amazon GameLift (p. 165) – Learn more about how to set up VPC peering.  

**Service API Reference:**  
- New APIs for VPC peering:  
  - CreateVpcPeeringAuthorization  
  - DescribeVpcPeeringAuthorizations  
  - DeleteVpcPeeringAuthorization  
  - CreateVpcPeeringConnection  
  - DescribeVpcPeeringConnections  
  - DeleteVpcPeeringConnection  
  - MatchmakingTicket  
  - EndTime and EstimatedWaitTime added.  
  - The documentation for DescribeMatchmaking was corrected. This operation can be used to retrieve a maximum of 10 matchmaking tickets. | **AWS SDK:** 2017-09-01  
Server SDK 3.1.7  
GameLiftLocal 1.0.0 |
<table>
<thead>
<tr>
<th>Date</th>
<th>Change</th>
<th>Documentation updates</th>
<th>API Versions</th>
</tr>
</thead>
</table>
| August 16, 2017 | **New Features:**<br>Amazon GameLift FlexMatch – Add matchmaking to your games using this customizable matchmaking service. With FlexMatch, you can design a rule set based on the team formats and player characteristics that best fit your game. | Developer Guide:  
- How Amazon GameLift FlexMatch Works (p. 13) – Get more information on FlexMatch key features and how it works.  
- Adding FlexMatch Matchmaking (p. 69) – Learn more about how to set up FlexMatch and customize it to your game. | AWS SDK: 2017-08-16  
Server SDK 3.1.7  
GameLiftLocal 1.0.0 |

**Server SDK:**
- Amazon GameLift Server API – New APIs for managing FlexMatch resources and starting new games with matchmaking.
### New Features:
- Amazon GameLiftMetrics are now supported in Amazon CloudWatch. This includes the ability to work with aggregated metrics for a group of fleets.
- Limit game session activations on instances in a fleet.

### Updates:
- Take advantage of additional metrics for automatic scaling.
- Use new console UI to set fleet scaling.

### Developer Guide:
- Monitoring Amazon GameLift (p. 178) – New monitoring section including list of metrics available in the Amazon GameLift console and in Cloudwatch.
- Deploy a GameLift Fleet for a Custom Game Build (p. 104) and Manage Fleet Records (p. 111) – Updated instructions on creating and updating fleet configurations.
- Auto-Scale Fleet Capacity (p. 123) – Updated instructions on setting manual and auto-scaling for a fleet.

### Service API Reference:
- New MetricGroups parameter added to enable aggregated metrics:
  - CreateFleet
  - UpdateFleetAttributes
  - FleetAttributes
  - RuntimeConfiguration – game session activation limits added.
<table>
<thead>
<tr>
<th>Date</th>
<th>Change</th>
<th>Documentation updates</th>
<th>API Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 11, 2017</td>
<td><strong>New Features:</strong></td>
<td>Developer Guide:</td>
<td>AWS SDK: 2017-04-11</td>
</tr>
<tr>
<td></td>
<td>• Amazon GameLift Local – Test your game integration locally.</td>
<td>• Testing Your Integration (p. 54) – New topic on setting up and using Amazon GameLift Local.</td>
<td>Server SDK 3.1.5</td>
</tr>
<tr>
<td></td>
<td>• Set a Queue policy that limits the latency allowed for individual players.</td>
<td>• Create a Queue (p. 133) – Updated topic on creating queues, with new information on player latency policies.</td>
<td>GameLiftLocal 1.0.0</td>
</tr>
<tr>
<td></td>
<td><strong>Updates:</strong></td>
<td>Service API Reference:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Changes to the Amazon GameLift Service API (part of the AWS SDK) to improve usability.</td>
<td>• PlayerLatencyPolicy (new data type)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PlacedPlayerSession (new data type)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• GameSessionPlacement – GameSessionId, IpAddress, Port, PlacedPlayerSessions added.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CreateGameSession – IdempotencyToken replaces GameSessionId.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• GameSessionQueue – GameSessionQueueArn added.</td>
<td></td>
</tr>
<tr>
<td>February 21, 2017</td>
<td><strong>New features:</strong></td>
<td>Developer Guide:</td>
<td>AWS SDK: 2017-02-21</td>
</tr>
<tr>
<td></td>
<td>• Multiple game engine support, including Unreal Engine, Unity, and custom C++ and C# game engines</td>
<td>• Game Architecture with Managed Amazon GameLift (p. 16) – Architecture diagram.</td>
<td>Server SDK 3.1.5</td>
</tr>
<tr>
<td></td>
<td>• Language support for the Server SDK is expanded to include C#</td>
<td>• Game Engines and Amazon GameLift (p. 34) – Help with using Amazon GameLift with various game engines, and plugin setup instructions for Unreal Engine and Unity.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• New game session creation using game session placements and cross-region queues</td>
<td>• Using Multi-Region Queues (p. 128) – Help with creating, managing, and tracking metrics for queues.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Custom player data support, with delivery directly to game server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>November 18, 2016</td>
<td><strong>New features:</strong></td>
<td>• Remote access to Amazon GameLift fleet instances</td>
<td>AWS SDK: 2016-11-18 Server SDK for C++: 3.1.0</td>
</tr>
<tr>
<td></td>
<td><strong>Developer Guide:</strong></td>
<td>• New topics:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Remotely Access Fleet Instances (p. 117) – How to get access and remotely connect to Amazon GameLift instances.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Debug Fleet Issues (p. 114) – Troubleshooting tips for new fleets that fail to activate.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Service API Reference:</strong></td>
<td>• For remote access:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• GetInstanceAccess (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• InstanceAccess (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• InstanceCredentials (new)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td></td>
<td>• Resource creation protection</td>
<td>• Revised topics:</td>
<td>Server SDK for C++: 3.1.0</td>
</tr>
<tr>
<td></td>
<td>• Access to instance data</td>
<td>• How Amazon GameLift Works (p. 3) – Added description of resource protection, and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Updates and corrections:</td>
<td>improved description of capacity handling.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Additional help for Linux.</td>
<td>• Added Linux-specific help:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Package Your Game Build Files (p. 92) – Install scripts for Linux.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upload a Custom Server Build to GameLift (p. 91) – New Linux examples.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deploy a GameLift Fleet for a Custom Game Build (p. 104) – New launch path</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>example for Linux.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Service API Reference:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CreateFleet and UpdateFleetAttributes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>– New</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ResourceCreationLimitPolicy parameter.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ResourceCreationLimitPolicy (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CreateGameSession – CreatorId added.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DescribeInstances (new)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>September 1, 2016</td>
<td>New features:</td>
<td>Developer Guide:</td>
<td>AWS SDK: 2016-09-01</td>
</tr>
<tr>
<td></td>
<td>• Game servers can now run on Linux</td>
<td>• New topics:</td>
<td>Server SDK for C++: 3.1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Amazon GameLift SDKs (p. 23) – Reference topic describing all Amazon GameLift SDKs,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>including supported languages and operating systems.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Service API Reference:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New OS parameters were added to the following:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• upload-build (p. 91) (CLI only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CreateBuild()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Build</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FleetAttributes</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>• Game session search</td>
<td>• New topics:</td>
<td>Server SDK for C++: 3.0.7</td>
</tr>
<tr>
<td></td>
<td>• Customized health checks</td>
<td>• Amazon GameLift Server API (C++) Reference (p. 232) – Complete reference documentation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Updates:</td>
<td>• Run Multiple Processes on a Fleet (p. 101) – Technical overview of capacity allocation and how to configure a fleet to run multiple processes.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Expanded support for capacity allocation (multiple processes per fleet instance)</td>
<td>• Tools and Resources (p. 26) – Comprehensive list of tools &amp; resources, including SDK version compatibility.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Amazon GameLift Server SDK for C++ now available for download</td>
<td>• Revised topics:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• All APIs for game client integration are now included in the AWS SDK.</td>
<td>• How Players Connect to Games (p. 12) – Expanded topic describes features related to game sessions, including the new search feature.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Add Amazon GameLift to Your Game Server (p. 42) – Integration steps have been revised for use with version 3.0.7 Server SDK for C++.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Add Amazon GameLift to Your Game Client (p. 47) – Integration steps have been revised for use with the AWS SDK for C++.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Service API Reference:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SearchGameSessions() (new)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>June 28, 2016</td>
<td>Updates:</td>
<td>• New server process health metrics</td>
<td>AWS SDK: 2016-06-28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Revised processes for fleet capacity allocation and game server launch settings</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Revised build packaging instructions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Developer Guide:</td>
<td>• Revised topics:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Package Your Game Build Files (p. 92) – Description now</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reflects how Amazon GameLift handles an install.bat file in a</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>game build.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Deploy a GameLift Fleet for a Custom Game Build (p. 104) – Instructions for</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>creating a fleet now cover capacity allocation using a runtime configuration.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• View Fleet Details (p. 172) and View Data on Game and Player Sessions (p. 176)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Console page descriptions now reflect current metrics and scaling tabs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Amazon GameLift and Game Client/Server Interactions (p. 51) – Descriptions and</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>diagram (p. 53) have been corrected to use callback function names from the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>samples, and to clarify that the onProcessTerminate() callback refers to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>shutting down a game server, not a game session.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Service API Reference:</td>
<td>• For new capacity allocation:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CreateFleet() – Runtime configuration added.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DescribeRuntimeConfiguration (new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UpdateRuntimeConfiguration (new)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-----------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For game server launch process:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• GameSession – Port number added.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PlayerSession – Port number added.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For health metrics:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FleetUtilization – New count added for active server processes.</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>March 10, 2016</td>
<td>New features:</td>
<td>Developer Guide:</td>
<td>AWS SDK 2016-03-10</td>
</tr>
<tr>
<td></td>
<td>• Auto-scaling</td>
<td>New topics:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Game session protection</td>
<td>• Auto-Scale Fleet Capacity (p. 123) – How to set up and manage auto-scaling policies.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Fleet capacity limits</td>
<td>• Manually Set Fleet Capacity (p. 122) – How to change the number of instances in a fleet and set limits.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• How Amazon GameLift Works (p. 3) – A technical overview of how Amazon GameLift manages game deployment across virtual resources.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Revised topics:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Deploy a GameLift Fleet for a Custom Game Build (p. 104) – Revised to include settings for game session protection and safe scaling.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Other changes:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Lumberyard-Amazon GameLift tutorial was moved to the GameDev Tutorials repository.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Service API Reference:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For auto-scaling:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• PutScalingPolicy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DescribeScalingPolicies</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DeleteScalingPolicy</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For game session protection:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DescribeGameSessionDetails</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CreateFleet (revised)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UpdateFleetAttributes (revised)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DescribeFleetAttributes (revised)</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Change</td>
<td>Documentation updates</td>
<td>API Versions</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------</td>
<td>--------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>February 9, 2016</td>
<td>Service launch</td>
<td>Developer Guide and API Reference for the Amazon GameLift service released on AWS.</td>
<td>AWS SDK 2016-02-09</td>
</tr>
</tbody>
</table>

- UpdateGameSession (revised)
- For fleet capacity limits:
  - UpdateFleetCapacity (revised)
  - DescribeFleetCapacity (revised)
AWS Glossary

For the latest AWS terminology, see the AWS Glossary in the AWS General Reference.