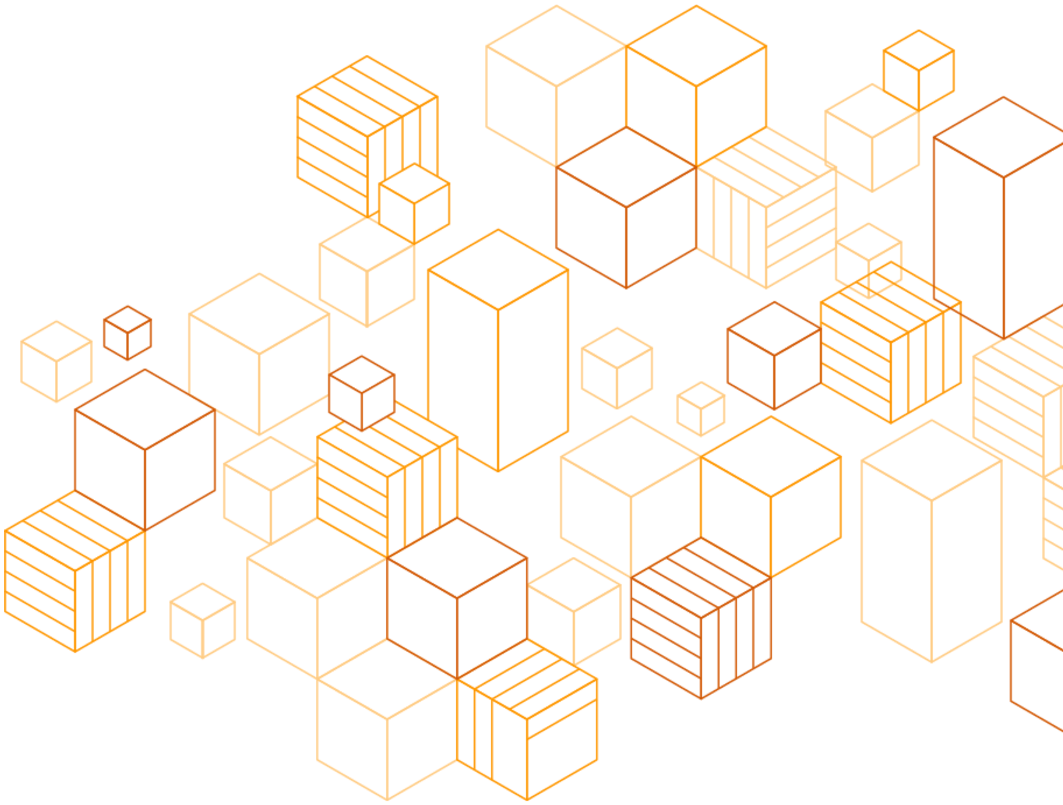
Introduction to Scalable Game Development Patterns on AWS

Second Edition

Published December 2019

Updated March 11, 2021



Notices

Contents

Getting started 1

Game design decisions.....1

Game client considerations.....3

Launching an initial game backend.....4

High availability, scalability, and security 8

Binary game data with Amazon S3 9

Expanding beyond AWS Elastic Beanstalk.....10

Reference architecture 11

Games as REST APIs.....13

HTTP load balancing.....14

HTTP automatic scaling.....18

Game servers 20

Matchmaking 21

Routing messages with Amazon SNS.....22

Last thoughts on game servers 23

Relational vs. NoSQL databases.....24

MySQL.....24

Amazon Aurora 27

Redis.....28

MongoDB.....28

Amazon DynamoDB.....29

Other NoSQL options.....32

Caching.....32

Binary game content with Amazon S3.....35

Content delivery and Amazon CloudFront 36

Uploading content to Amazon S3 37

Amazon S3 performance considerations 42
Loosely coupled architectures with asynchronous jobs 44
Leaderboards and avatars 44
Amazon SQS 45
Other queue options 47
Cost of the cloud 47
Conclusion and next steps 48
Contributors 49
Further reading 49
Document revisions 50

Introduction

Whether you’re an up-and-coming mobile developer or an established AAA game studio, you understand the challenges involved with launching a successful game in the current games landscape. Not only must the game be compelling, but users also expect a wide range of online features such as friend lists, leaderboards, weekly challenges, various multiplayer modes, and ongoing content releases.

To successfully execute a game launch, it’s critical to get favorable app store ratings and reviews on popular e-retail channels to provide sales and awareness momentum for your game—like the first weekend of a movie release. To deliver these features, you need a server backend. The server backend can consist of both the actual game servers for multiplayer games or servers that power the game services such as chat, matchmaking, and so on. The server backend must be able to scale up at a moment’s notice, in the event that the game goes viral and suddenly explodes from 100 to 100,000 users. At the same time, the backend must be cost effective, so that you don’t overpay for unused server capacity.

Amazon Web Services (AWS) is a flexible, cost-effective, easy-to-use cloud service. By running your game on AWS, you can leverage capacity on demand to scale up and down with your users, rather than having to guess at your server demands and potentially over-purchase or under-purchase hardware. Many indie, mobile, and AAA developers have recognized the advantages of AWS and are having success running their games on the AWS Cloud.

This book is broken into sections covering the different features of modern games, such as friend lists, leaderboards, game servers, messaging, and user-generated content. You can start small and just use the AWS components and services you need. As your game evolves and grows, you can revisit this book and evaluate additional AWS features.

Getting started

If you are just getting started developing your game, it can be challenging to figure out where to begin with your backend server development. Thankfully, AWS can help you get started quickly, because you don't have to make a decision about every service that you're going to use up front. As you iterate on your game, you can add AWS services over time. This approach enables you to develop additional game features or backend functionality without having to plan for everything at the beginning. We encourage you to start based on the game features that you need, and then add more AWS features as your game evolves. In this section, we'll look at some common game features that determine which types of services you'll need.

Game design decisions

Modern social, mobile, and AAA games tend to share the following common tenets that affect server architecture:

- **Pick up and play anywhere –** Players expect their saved games, profiles and other data to be stored online to allow the easily move from device to device. This operation typically involves synchronizing and merging local data as you move from one device to another, so a simple data storage solution is not always the right solution.
- **Leaderboards and rankings –** Players continue to look for a competitive experience similar to classic arcade games. Increasingly, though, the focus is on friends' leaderboards, rather than just a single global high score list. This requires a more sophisticated leaderboard that can sort in multiple dimensions, while maintaining good performance.
- **Free-to-play –** One of the biggest shifts over the past few years has been the widespread move to free-to-play. In this model, games are free to download and play, and the game earns money through in-app purchases for items such as weapons, outfits, power-ups, and boost points, as well as advertising. The game is funded by a small minority of users that purchase these items, with the vast majority of users playing for free. This means that your game backend must be as cost-effective as possible, and must be able to scale up and down as needed. Even for premiere AAA games, larger percentages of revenue are now coming from content updates and in-game purchases.

- **Analytics –** Maximizing long-tail revenue requires that games collect and analyze a large number of metrics regarding gameplay patterns, favorite items, purchase preferences, and so forth. Ensuring that new game features target those areas of the game where users are spending their time and money is a critical factor in the success of in-game purchases.
- **Content updates –** Games that achieve the highest player retention tend to have a continuous release cycle of new items, levels, challenges, and achievements. The continuing trend of games becoming more of a service that a single product reinforces the need for constant post launch changes. These features require frequent updates with new data and game assets. By using a content delivery network (CDN) to distribute game content, you can cut costs and increase download speed.
- **Asynchronous gameplay –** Although larger games generally include a real-time online multiplayer mode, games of all kinds are realizing the importance of asynchronous features to keep players engaged. Examples of asynchronous play include competing against your friends based on points, unlocks, badges, or similar achievements. This type of game play gives players the feel of a connected game experience, even if they aren’t online all the time, or if they are using slower networks like 3G or 4G for mobile games.
- **Push notifications –** A common method of getting users to come back to the game is to send targeted push notifications to their mobile device. For example, a user might get a notification that their friend beat their score, or that a new challenge or level is available. This draws the user back into the core game experience even when they’re not directly playing.
- **Unpredictable clients –** Modern games run on a wide variety of platforms including mobile devices, consoles, PCs, and browsers. One user could be roaming on their portable device, playing against a console user on Wi-Fi, and both would expect a consistent experience. For this reason, it’s necessary to leverage stateless protocols (for example, HTTP) and asynchronous calls as much as possible.

Each of these game features has an impact on your server features and technology. For example, if you have a simple Top 10 leaderboard, you may be able to store it in a single MySQL or Amazon Aurora database table. However, if you have complex leaderboards with multiple sort dimensions, it may be necessary to use a NoSQL option such as Amazon ElastiCache or Amazon DynamoDB (discussed later in this book).

Game client considerations

Although the focus of this book is on the architecture you can deploy on AWS, the implementation of your game client can also have an impact on your game's scalability. It also affects how much your game backend costs to run because frequent network requests from the client use more bandwidth and require more server resources. Here are a few important guidelines to follow:

- **All network calls should be asynchronous and non-blocking.** This means that when a network request is initiated, the game client continues on, without waiting for a response from the server. When the server responds, this triggers an event on the client, which is handled by a callback of some kind in the client code. On iOS, AFNetworking is one popular approach. Browser games should use a call such as jQuery.ajax() or the equivalent, and C++ clients should consider libcurl, std::async or similar libraries. Similarly, popular game engines usually include an asynchronous method for network and web requests. For example, Unity offers UnityWebRequest and Unreal Engine has HttpRequest.
- **Use JSON to transport data.** It's compact, cross-platform, fast to parse, has lots of library support, and contains data type information. If you have large payloads, simply gzip them, because the majority of web servers and mobile clients have native support for gzip. Don't waste time over-optimizing—any payload in the range of hundreds of kilobytes should be adequate. We have also seen developers use Apache Avro and MessagePack depending on their use case, comfort level with the formats, and availability of libraries. **Note:** An exception to this rule is multiplayer gameplay packets, which are typically UDP.
- **Use HTTP/1.1 with Keepalives, and reuse HTTP connections between requests.** This minimizes the overhead your game incurs when making network requests. Each time you have to open a new HTTP socket, this requires a three-way TCP handshake, which can add upwards of 50 milliseconds (ms). In addition, repeatedly opening and closing TCP connections will accumulate large numbers of sockets in the TIME_WAIT state on your server, which consumes valuable server resources.
- **Always POST any important data from the client to the server over SSL.** This includes login, stats, save data, unlocks, and purchases. The same applies for any GET, PUT, and DELETE requests because modern computers are efficient at handling SSL and the overhead is low. AWS enables you to have our Elastic Load Balancer handle the SSL workload, which completely offloads it from your servers.

- **Never store security-critical data such as AWS access keys or other tokens on the client device**, either as part of your game data or user data. Access key IDs and secret access keys allow the possessors of those keys to make programmatic calls to AWS from the AWS Command Line Interface (AWS CLI), AWS Tools for Windows PowerShell, the AWS SDKs, or direct HTTP calls using the APIs for individual AWS services. If somebody roots or jailbreaks their device, you risk the possibility that they could gain access to your server code, user data, and even your AWS billing account. In the case of PC games, your keys likely exist in memory when the game client is running, and pulling them out isn't that hard for someone with the know-how. You have to assume anything you store on a game client will be compromised. If you want your game client to directly access AWS services, consider using Amazon Cognito Federated Identities which allows your application to obtain temporary, limited-privilege credentials.
- **As a precaution you should never trust what a game client sends you.** It's an untrusted source and you should always validate what you receive. Sometimes it's malicious traffic (SQL Injection, XSS, etc.), but sometimes it can be something as trivial as someone having their device clock set to a time that's in the past.

Many of these concerns are not specific to AWS and are typical client/server safety issues, but keeping them in mind will help you design a game that performs well and is reasonably secure.

Launching an initial game backend

With the previous game features and client considerations in mind, let's look at a strategy for getting an initial game backend up and running on AWS as quickly as possible. We'll make use of a few key AWS services, with the ability to add more as the game evolves.

To ensure we're able to scale out as our game grows in popularity, we'll leverage stateless protocols as much as possible. Creating an HTTP/JSON API for the bulk of our game features allows us to add instances dynamically and easily recover from transient network issues. Our game backend consists of a server that talks HTTP/JSON, stores data in MySQL, and uses Amazon Simple Storage Service (Amazon S3) for binary content. This type of backend is easy to develop and can scale effectively.

A common pattern for game developers is to run a web server locally on a laptop or desktop for development, and then push the server code to the cloud when it's time to deploy. If you follow this pattern, [AWS Elastic Beanstalk]() can greatly simplify the process of deploying your code to AWS.

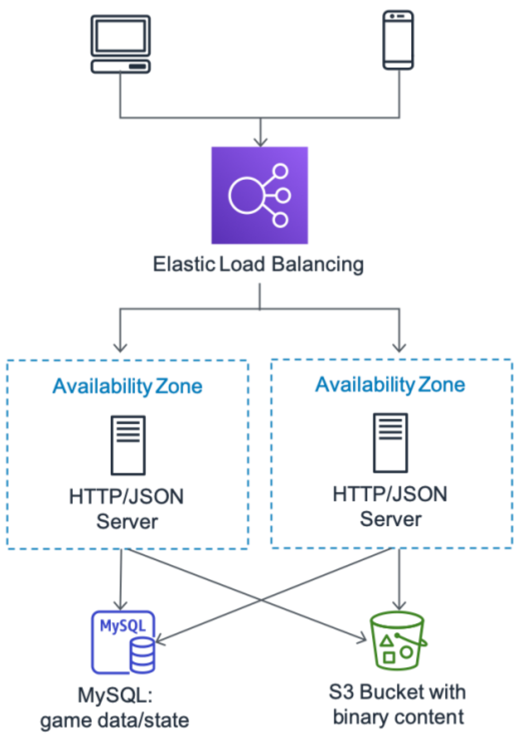


Figure 1: A high level overview of your first game backend running on AWS

Elastic Beanstalk is a deployment management service that sits on top of other AWS services such as Amazon Elastic Compute Cloud (Amazon EC2), Elastic Load Balancing, and Amazon Relational Database Services (Amazon RDS). Amazon EC2 is a web service that provides secure, resizable compute capacity in the cloud. It is designed to make at-scale cloud computing easier for developers. The Amazon EC2 simple web service interface allows you to obtain and configure computing capacity with minimal friction. It reduces the time required to obtain and boot new server instances to minutes, which allows you to quickly scale capacity (up or down) as your computing requirements change.

Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances. It enables you to achieve fault tolerance in your applications. Elastic Load Balancing offers three types of load balancers that feature high availability, automatic scaling, and robust security. These are the Application Load Balancer that routes traffic based on advanced application-level information that includes the content of the request and is most suited to HTTP and HTTPS traffic, the Network Load Balancer that is best suited for TCP, UPD and TLS traffic, and the Classic Load Balancer that works with the EC2-classic network. The Classic Load Balancer is ideal for simple load balancing of traffic across multiple EC2 instances. The Application Load Balancer is ideal for applications that need advanced routing capabilities, microservices, and container-based architectures. The Network Load Balancer would be ideal for routing messages to persistent game servers, chat services and other stateful servers.

Amazon RDS makes it easy to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching, and backups. Amazon RDS supports many familiar database engines, including Amazon Aurora, PostgreSQL, MySQL, and more.

You can push a zip, war, or git repository of server code to Elastic Beanstalk. Elastic Beanstalk takes care of launching EC2 server instances, attaching a load balancer, setting up Amazon CloudWatch monitoring alerts, and deploying your application to the cloud. In short, Elastic Beanstalk can set up most of the architecture shown in *Figure 1* automatically.

To see Elastic Beanstalk in action, log in to the AWS Management Console and follow the Getting Started Using Elastic Beanstalk tutorial to create a new environment with the programming language of your choice. This will launch the sample application and boot a default configuration. You can use this environment to get a feel for the Elastic Beanstalk control panel, how to update code, and how to modify environment settings. If you’re new to AWS, you can use the AWS Free Tier to set up these sample environments.

Note: The sample production environment described in this book will incur costs because it includes AWS resources that aren’t covered under the free tier.

With the sample application up, let’s create a new Elastic Beanstalk application for our game, and two new environments, one for development and one for production. We’ll customize these a bit for our game. Use the following table to determine which settings to change, depending on the environment type. For detailed instructions, see Managing and Configuring AWS Elastic Beanstalk Applications and then follow the instructions for Creating an AWS Elastic Beanstalk Environment in the AWS Elastic Beanstalk Developer Guide.

Note: In the following table, replace My Game and mygame values with the name of your game.

Table 1: Configuration settings for gaming environments

Configuration Setting	Development Value	Production Value
Application Name	My Game	My Game
Environment Name	mygame-dev	mygame-prod
Instance Type	t2.micro	M5.large
Create RDS DB instance?	Yes	Yes
DB Engine	Mysql	** Not recommended
Instance Class	db.t2.micro	N/A
Allocated Storage	5 GB	N/A

By using two environments, you can enable a simple and effective workflow. As you integrate new game backend features, you push your updated code to the development environment. This triggers Elastic Beanstalk to restart the environment and create a new version. In your game client code, create two configurations, one that points to development and one that points to production. Use the development configuration to test your game, and then use the production profile when you want to create a new game version to publish to the appropriate app stores.

When your new game client is ready for release, choose the correct server code version from the development environment, and deploy it to the production environment. By default, deployments incur a brief period of downtime while your app is being updated and restarted. To avoid downtime for production deployments, you can follow a pattern known as swapping URLs or blue/green deployment. In this pattern, you deploy to a standby production environment, and then update DNS to point to the new environment. For more details on this approach, see Blue/Green Deployments with AWS Elastic Beanstalk in the AWS Elastic Beanstalk Developer Guide.

Important: We don't recommend that you use Elastic Beanstalk to manage your database in a production environment because this ties the lifecycle of the database instance (DB instance) to the lifecycle of your application's environment.

Instead, we recommend that you run a DB instance in Amazon Aurora and configure your application to connect to it on launch. You can also store connection information in Amazon S3 and configure Elastic Beanstalk to retrieve that information during deployment with .ebextensions. You can add AWS Elastic Beanstalk configuration files (.ebextensions) to your web application's source code to configure your environment and customize the AWS resources that it contains. Configuration files are YAML formatted documents with a .config file extension that you place in a folder named .ebextensions and deploy in your application source bundle.

For more information, see Advanced Environment Customization with Configuration Files (.ebextensions) in the AWS Elastic Beanstalk Developer Guide.

High availability, scalability, and security

For the production environment, you need to ensure that your game backend is deployed in a fault-tolerant manner. Amazon EC2 is hosted in multiple AWS Regions worldwide. You should choose a Region that is near the bulk of your game's customers. This ensures that your users have a low-latency experience with your game. For more information and a list of the latest AWS Regions, see the AWS Global Infrastructure webpage.

Within each Region are multiple, isolated locations known as Availability Zones, which you can think of as logical data centers. Each of the Availability Zones within a given Region is isolated physically, yet connected via high-speed networking so they can be used together. Balancing your servers across two or more Availability Zones within a Region is a simple way to increase your game's high availability. Using two Availability Zones is a good balance of reliability and cost for most games, since you can pair your server instances, database instances, and cache instances together.

Elastic Beanstalk can automatically deploy across multiple Availability Zones for you. To use multiple Availability Zones with Elastic Beanstalk, see Auto Scaling Group for Your Elastic Beanstalk Environment in the AWS Elastic Beanstalk Developer Guide. For additional scalability, you can use automatic scaling to add and remove instances from these Availability Zones. For best results, consider modifying the automatic scaling trigger to specify a metric (such as CPU usage) and threshold based on your application's performance profile. If the threshold you specify is hit, Elastic Beanstalk automatically launches additional instances. This is covered in more detail in the HTTP Automatic Scaling section of this book.

For development and test environments, a single Availability Zone is usually adequate so you can keep costs low—assuming you can tolerate a bit of downtime in the event of a failure. However, if your development environment is actually used by QA testers to validate builds late at night, you probably want to treat this more like a production environment. In that case, leverage multiple Availability Zones like you would in production.

Finally, set up the load balancer to handle SSL termination, so that SSL encryption and decryption is offloaded from your game backend servers. This is covered in Configuring HTTPS for Your Elastic Beanstalk Environment in the AWS Elastic Beanstalk Developer Guide. For security reasons, we strongly recommend that you use SSL for your game backend. For more Elastic Load Balancing tips, see the HTTP Load Balancing section of this book.

Binary game data with Amazon S3

Next, you'll need to create an S3 bucket for each Elastic Beanstalk server environment that you created previously. This S3 bucket stores your binary game content, such as patches, levels, and assets. Amazon S3 uses an HTTP-based API for uploading and downloading data, which means that your game client can use the same HTTP library for talking to your game servers that’s used to download game assets. With Amazon S3, you pay for the amount of data you store and the bandwidth for clients to download it. For more information, see Amazon S3 Pricing.

To get started, create an S3 bucket in the same Region as your servers. For example, if you deployed Elastic Beanstalk to the us-west-2 (Oregon) Region, choose this same Region for Amazon S3. For simplicity, and because S3 requires bucket names to be unique across all S3, use a similar naming convention for the bucket that you used for your Elastic Beanstalk environment (for example, mygame-dev or mygame-prod) along with other unique identification like com.mycompany.mygame-dev. For step-by-step directions, see Create a Bucket in the Amazon Simple Storage Service Getting Started Guide. Remember to create a separate S3 bucket for each of your Elastic Beanstalk environments (that is, development, production, etc.)

By default, S3 buckets are private, and require that users authenticate to download content for security. For game content, you have two options. You could make the bucket public, which means that anyone with the bucket name can download your game content, but this is not recommended. However, a better way to manage authentication is to use signed URLs, which is a feature that enables you to pass Amazon S3 credentials as part of the URL. In this scheme, your game server code redirects users to an Amazon S3 signed URL, which you can set to expire after a period of time. For instructions on how to create a signed URL, see Authenticating Requests (AWS Signature Version 4) in the Amazon S3 API Reference. If you are using one of the official AWS SDKs with your game server, there is also a good chance that the SDK has built-in methods for generating a pre-signed URL. A pre-signed URL gives you access to the object identified in the URL, provided that the creator of the pre-signed URL has permissions to access that object. Generating a pre-signed URL is a completely offline operation (no API calls are involved), making it a very fast operation.

Finally, as your game grows, you can use Amazon CloudFront, a content delivery network (CDN), to provide better performance and save you money on data transfer costs. For more information, see What is Amazon CloudFront in the Amazon CloudFront Developer Guide.

Expanding beyond AWS Elastic Beanstalk

As your game increases in popularity, your core game backend must scale and respond to demand over a period of time. By using HTTP for the bulk of your calls, you are able to easily scale up and down in response to changing usage patterns. Storing binary data in Amazon S3 saves you money compared to serving files from Amazon EC2, and Amazon S3 also takes care of data availability and durability for you. Amazon RDS provides you with a managed MySQL database that you can grow over time with Amazon RDS features, such as read replicas.

If your game needs additional functionality, you can easily expand beyond Elastic Beanstalk to other AWS services, without having to start over. Elastic Beanstalk supports configuring other AWS services via the Elastic Beanstalk Environment Resources. For example, you can add a caching tier using Amazon ElastiCache, which is a managed cache service that supports both Memcached and Redis. For details about adding an ElastiCache cluster, see the Example: ElastiCache in the AWS Elastic Beanstalk Developer Guide.

Of course, you can always just launch other AWS services yourself and then configure your app to use them. For example, you could choose to augment or even replace your RDS MySQL DB instance with Amazon Aurora Serverless, an on-demand, automatic scaling SQL database or, Amazon DynamoDB, the AWS managed NoSQL offering. Even though we're using Elastic Beanstalk to get started, you still have access to all other AWS services as your game grows.

Reference architecture

With our core game backend up and running, the next step is to examine the other AWS services that could be useful for our game. Before continuing, let’s look at the following reference architecture for a horizontally scalable game backend. This diagram depicts a game backend that supports a wide set of game features, including login, leaderboards, challenges, chat, binary game data, user-generated content, analytics, and online multiplayer. Not all games have all these components, but this diagram provides a good visualization of how they would all fit together. In the remaining sections of this book, we’ll cover each component in detail.

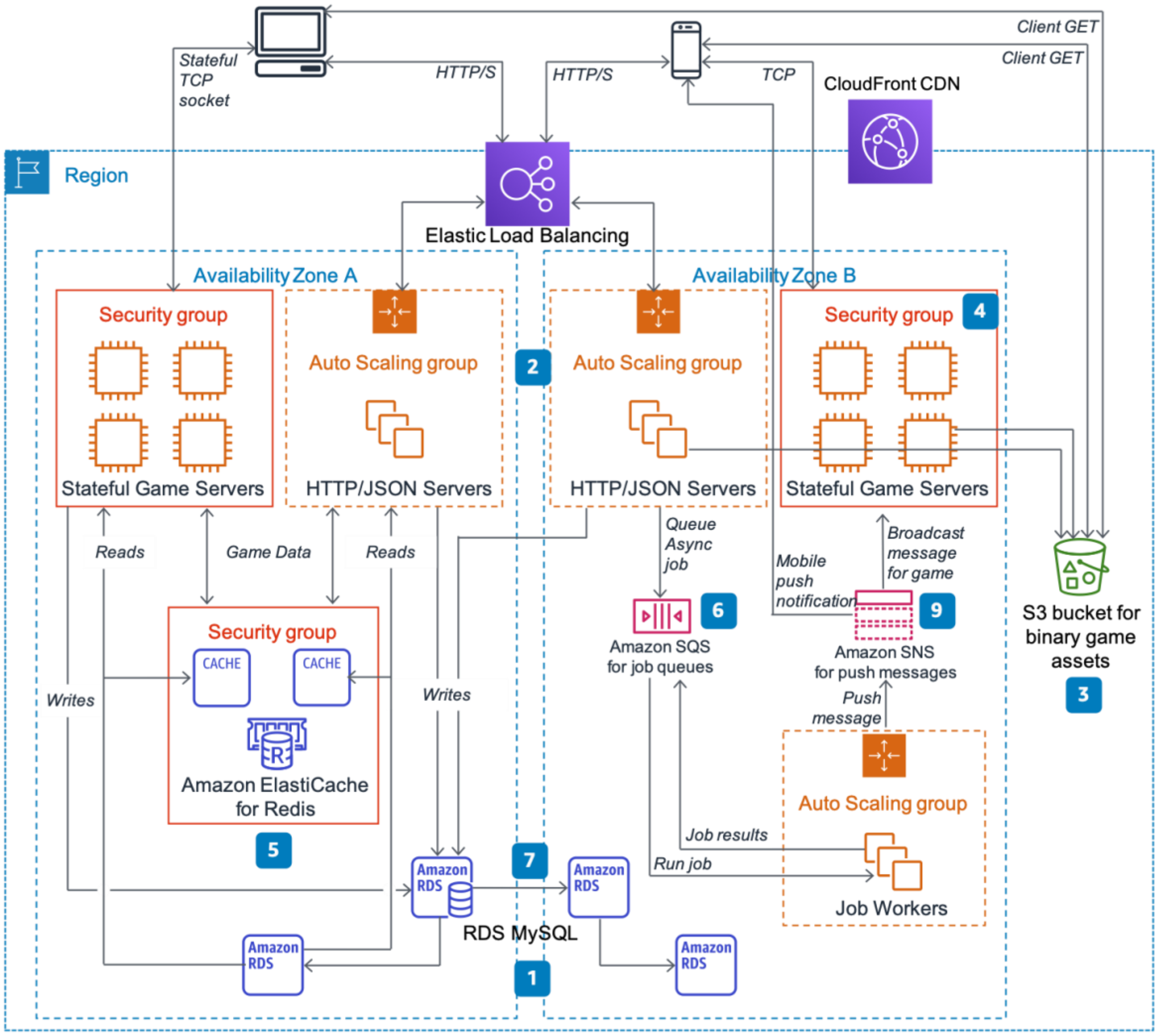


Figure 2: A fully production-ready game backend running on AWS

Figure 2 may seem overwhelming at first, but it's really just an evolution of the initial game backend we launched using Elastic Beanstalk. The following table explains numbered areas of the diagram.

Table 2: Reference architecture callouts

Callout	Description
1	The diagram shows two Availability Zones set up with identical functionality for redundancy. Not all components are shown in both Availability Zones due to space constraints, but both Availability Zones function equivalently. These Availability Zones can be the same as the two Availability Zones you initially chose using Elastic Beanstalk.
2	The HTTP/JSON servers and master/slave DBs can be the same ones you launched using Elastic Beanstalk. You continue to build out as much of your game functionality in the HTTP/JSON layer as possible. You can use HTTP automatic scaling to add and remove EC2 HTTP instances automatically in response to user demand. For more information, see the HTTP Automatic Scaling section of this book.
3	You can use the same S3 bucket that you initially created for binary data. Amazon S3 is built to be highly scalable and needs little tuning over time. As your game assets and user traffic continues to expand, you can add Amazon CloudFront in front of S3 to boost download performance and save costs.
4	If your game has features requiring stateful sockets, such as chat or multiplayer gameplay, these features are typically handled by game servers running code just for those features. These servers run on EC2 instances separate from your HTTP instances. For more information, see the Game Servers section of this book.
5	As your game grows and your database load increases, the next step is to add caching, typically by using Amazon ElastiCache, which is the AWS managed caching service. Caching frequently accessed items in ElastiCache offloads read queries from your database. This is covered in the Caching section of this book.
6	The next step is to look at moving some of your server tasks to asynchronous jobs, and using Amazon Simple Queue Service (Amazon SQS) to coordinate this work. This allows for a loosely coupled architecture where two or more components exist and each has little or no knowledge of other participating components, but they interoperate to achieve a specific purpose. Amazon SQS eliminates dependencies on the other components in a loosely coupled system. For example, if your game allows users to upload and share assets such as photos or custom characters, you should execute time-intensive tasks such as image resizing in a background job. This results in quicker response times for your game, while also decreasing the load on your HTTP server instances. These strategies are discussed in the Loosely Coupled Architectures with Asynchronous Jobs section of this book.

Callout	Description
7	As your database load continues to grow, you can add Amazon RDS read replicas to help you scale out your database reads even further. This also helps reduce the load on your main database because you can read from the replica and you only access the master database to write. This is covered in the Relational vs. NoSQL Databases section of this book.
8 (Not Shown)	At some point, you may decide to introduce a NoSQL service such as Amazon DynamoDB to supplement your main database for functionality such as leaderboards, or to take advantage of NoSQL features such as atomic counters. We discuss these options in the Relational vs. NoSQL Databases section .
9	If your game includes push notifications, you can use Amazon Simple Notification Service (Amazon SNS) and its support for Mobile Push to simplify the process of sending push messages across multiple mobile platforms. Your EC2 instances can also receive Amazon SNS messages, which enables you to do things like broadcast messages to all players currently connected to your game servers.

If you look at a single Availability Zone in Figure 2 and compare it to the core game backend we launched with Elastic Beanstalk, you can see how scaling your game builds on the initial backend pieces by adding caching, database replicas, and background jobs. With this in mind, let’s look at each component.

Games as REST APIs

As mentioned earlier, to make use of horizontal scalability, you should implement most of your game’s features using an HTTP/JSON API, which typically follows the REST architectural pattern. Game clients, whether on mobile devices, tablets, PCs, or consoles, make HTTP requests to your servers for data such as login, sessions, friends, leaderboards, and trophies. Clients do not maintain long-lived connections to the server, which makes it easy to scale horizontally by adding HTTP server instances. Clients can recover from network issues by simply retrying the HTTP request.

When properly designed, a REST API can scale to hundreds of thousands of concurrent players. This is the pattern we followed in the previous Elastic Beanstalk example. RESTful servers are straightforward to deploy on AWS, and they benefit from the wide variety of HTTP development, debugging, and analysis tools that are available on AWS.

Nevertheless, some modes of gameplay benefit from a stateful two-way socket that can receive server-initiated messages. Examples include real-time online multiplayer, chat, or game invites. If your game doesn’t have these features, you can implement all of your functionality using a REST API. We'll discuss stateful servers later in this book, but first let's focus on our REST layer.

Deploying a REST layer to Amazon EC2 typically consists of an HTTP server such as Nginx or Apache, plus a language-specific application server. The following table lists some of the popular packages that game developers use to build REST APIs.

Table 3: Packages to build REST APIs

Language	Package
Node.js	Express, Restify, Sails
Python	Eve, Flask Bottle
Java	Spring, Jersey
Go	Gorilla Mux, Gin
PHP	Slim, Silex
Ruby	Rails, Sinatra, Grape

This is just a sampling–you can build a REST API in any web-friendly programming language. Since Amazon EC2 gives you complete root access to the instance, you can deploy any of these packages. For Elastic Beanstalk, there are some restrictions on supported packages. For details, see the Elastic Beanstalk FAQs.

RESTful servers benefit from medium-sized instances, since this enables more to be deployed horizontally at the same price point. Medium-sized instances from the general purpose instance family (for example, M5) or compute optimized instance family (for example, C5) are a good match for REST servers.

HTTP load balancing

Load balancing RESTful servers is very straightforward because HTTP connections are stateless. AWS offers Elastic Load Balancing, which is the easiest approach to HTTP load balancing for games on Amazon EC2. You may recall from our example game backend that Elastic Beanstalk automatically deploys an Elastic Load Balancing load balancer to load balance your EC2 instances for you. If you use Elastic Beanstalk to get started, you will already have an Elastic Load Balancing load balancer running.

Follow these guidelines to get the most out of Elastic Load Balancing:

- Always configure Elastic Load Balancing to balance between at least two Availability Zones for redundancy and fault tolerance. Elastic Load Balancing handles balancing traffic between the EC2 instances in the Availability Zones that you specify. If you want an equal distribution of traffic on servers, you should also enable cross-zone load balancing even if there are an unequal number of servers per Availability Zone. This ensures optimal usage of servers in your fleet.
- Configure Elastic Load Balancing to handle SSL encryption and decryption. This offloads SSL from your HTTP servers, which means that there is more CPU for your application code. For more information, see Create an HTTPS Load Balancer in the Classic Load Balancer Guide. To test SSL for development purposes, see How to Create a Self-Signed SSL Certificate in the AWS Certificate Manager User Guide.
- Elastic Load Balancing automatically removes any EC2 instances that fail from its load balancing pool. To ensure that the health of your HTTP EC2 instances is accurately monitored, configure your load balancer with a custom health check URL. Then, write server code that responds to that URL and performs a check on your application’s health. For example, you could set up a simple health check that verifies that you have DB connectivity. The health check returns `200 Ok` if your health checks pass or `500 Server Error` if your instance is unhealthy.
- Each Elastic Load Balancing load balancer that you deploy must have a unique DNS name. To set up a custom DNS name for your game, you can use a DNS alias (CNAME) to point your game’s domain name to the load balancer. For detailed instructions, see Configure a Custom Domain Name for Your Classic Load Balancer in the Elastic Load Balancing Guide. Note that when your load balancer scales up or down, the IP addresses that the load balancer uses change—make sure you are using a DNS CNAME alias to the load balancer and that you’re not referencing the load balancer’s current IP addresses in your DNS domain.

- Elastic Load Balancing is designed to scale up by roughly a factor of 50 percent every 5 minutes. For the vast majority of games, this works well, even when they suddenly go viral. However, if you are anticipating a sudden huge spike in traffic—perhaps due to a new downloadable content release or marketing promotion—Elastic Load Balancing can be pre-warmed to scale up in advance for this event. To pre-warm Elastic Load Balancing, submit an AWS support request with the anticipated load (this requires at least Business Level Support). For more details on Elastic Load Balancing pre-warming and best practices for running load tests against Elastic Load Balancing, see the AWS article Best Practices in Evaluating Elastic Load Balancing.

Application Load Balancer

Application Load Balancer is the second generation load balancer that provides more granular control over traffic routing based at the HTTP/HTTPS layer. In addition to the features described in the previous section, the following features that come with Application Load Balancer can be highly beneficial to a gaming centric workload:

- Explicit support for Amazon EC2 Container Service (Amazon ECS) – Application Load Balancer can be configured to load balance containers across multiple ports on a single EC2 instance. Dynamic ports can be specified in an ECS task definition, which will give the container an unused port when scheduled on EC2 instances.
- HTTP/2 support – A revised edition of the older HTTP/1.1 protocol, HTTP/2 and Application Load Balancer together deliver additional network performance as a binary protocol, as opposed to a textual one. Binary protocols are inherently more efficient to process and are much less error prone, which can improve stability. Additionally, HTTP/2 supports multiplexing, which enables the reuse of TCP connections for downloading content from multiple origins and cuts down on network overhead.
- Native IPv6 support – With the near exhaustion of IPv4 addresses, many application providers are changing to a model where applications without IPv6 support are rejected on their services. Application Load Balancer natively supports IPv6 endpoints and routing to VPC IPv6 addresses.

- WebSockets support – Like HTTP/2, Application Load Balancer supports the WebSocket protocol, which enables you to set up a longstanding TCP connection between a client and server. This is a much more efficient method than standard HTTP connections, which were usually held open with a sort of heartbeat, which contributes to network traffic. WebSocket is a great use case for delivering dynamic data like updated leaderboards while minimizing traffic and power use on a mobile device. Elastic Load Balancing enables the support of WebSockets by changing the listener from HTTP to TCP. However, when it's in TCP Mode, Elastic Load Balancing allows the Upgrade header when a connection is established, and then the Elastic Load Balancing load balancer terminates any connection that is idle for more than 60 seconds (for example, a packet isn't sent within that timeframe). This means that the client has to reestablish the connection and any WebSocket negotiation fails if the Elastic Load Balancing load balancer sends an upgrade request and establishes a WebSocket connection to other backend instances.

Custom load balancer

Alternatively, you can deploy your own load balancer to Amazon EC2, if you need specific features or metrics that Elastic Load Balancing does not provide. Popular choices for games include HAProxy and F5's BIG-IP Virtual Edition, both of which can run on Amazon EC2. If you decide to use a custom load balancer, follow these recommendations:

- Deploy the load balancer software (such as HAProxy) to a pair of EC2 instances, each in a different Availability Zone for redundancy.
- Assign an Elastic IP address to each instance. Create a DNS record containing both of those Elastic IP addresses as your entry point. This allows DNS to round robin between your load balancer instances.
- If you are using Amazon Route 53, our highly available and scalable cloud Domain Name System (DNS) web service, use Route 53 health checks to monitor your load balancer EC2 instances to detect failure. This ensures that traffic doesn't get routed to a load balancer that is down.
- If you want HAProxy to handle SSL traffic, use the latest development version of HAProxy 1.5 or later.

- If you decide to deploy your own load balancer, keep in mind that there are several aspects you need to handle on your own. First and foremost, if your load surpasses what your load balancer instances can handle, you need to launch additional EC2 instances and follow the previous steps to add them to your application stack. In addition, new auto-scaled application instances aren't automatically registered with your load balancer instances. You need to write a script that updates the load balancer configuration files and restarts the load balancers.

If you are interested in HAProxy as a managed service, consider AWS OpsWorks, which uses Chef Automate to manage EC2 instances and can deploy HAProxy as an alternative to Elastic Load Balancing.

HTTP automatic scaling

The ability to dynamically grow and shrink server resources in response to user patterns is a primary benefit of running on AWS. Automatic scaling enables you to scale the number of EC2 instances in one or more Availability Zones, based on system metrics such as CPU utilization or network throughput. For an overview of the functionality that Amazon EC2 Auto Scaling provides, see What Is Amazon EC2 AutoScaling? and then walk through Getting Started with Amazon EC2 Auto Scaling.

You can use Amazon EC2 Auto Scaling with any type of EC2 instance, including HTTP, a game server, or a background worker. HTTP servers are the easiest to scale because they sit behind a load balancer that distributes requests across server instances. Auto Scaling handles the registration or deregistration of HTTPbased instances from Elastic Load Balancing dynamically, which means that traffic will be routed to a new instance as soon as it's available.

To use automatic scaling effectively, choose appropriate metrics to trigger scale up and scale down activities. To determine your metrics, follow these guidelines:

- `CPUUtilization` is often a good Amazon CloudWatch metric to use. Web servers tend to be CPU limited, whereas memory remains fairly constant when the server processes are running. A higher percentage of CPU tends to show that the server is becoming overloaded with requests. For finer granularity, pair `CPUUtilization` with `NetworkIn` or `NetworkOut`.

- Benchmark your servers to determine good values to scale on. For HTTP servers, you can use a tool such as Apache Bench or HTTPerf to measure your server response times. Increase the load on your servers while monitoring CPU or other metrics. Make note of the point at which your server response times degrade, and see how this correlates to your system metrics.
- When configuring your Amazon EC2 Auto Scaling group, choose two Availability Zones, and a minimum of two servers. This ensures your game server instances are properly distributed across multiple Availability Zones for high availability. Elastic Load Balancing takes care of balancing the load between multiple Availability Zones for you.

For details on configuring scaling policies, see Dynamic Scaling for Amazon EC2 Auto Scaling in the Amazon EC2 Auto Scaling User Guide.

Installing application code

When you use automatic scaling with Elastic Beanstalk, Elastic Beanstalk takes care of installing your application code on new EC2 instances as they're scaled up. This is one of the advantages of the managed container that Elastic Beanstalk provides.

However, if you're using automatic scaling without Elastic Beanstalk, you need to take care of getting your application code onto your EC2 instances to implement automatic scaling. If you are already using Chef or Puppet, consider using them to deploy application code on your instances. AWS OpsWorks automatic scaling, which uses Chef to configure instances, provides both time-based and load-based automatic scaling. With OpsWorks, you can also set up custom startup and shutdown steps for your instances as they scale. OpsWorks is a great alternative to managing automatic scaling if you're already using Chef, or if you're interested in using Chef to manage your AWS resources. For more information, see Managing Load with Time-based and Load-based Instances in the AWS OpsWorks User Guide.

If you're not using any of these packages, you can use the Ubuntu CloudInit package as a simple way to pass shell commands directly to EC2 instances. You can use `cloud-init` to run a simple shell script that fetches the latest application code and starts up the appropriate services. This is supported by the official Amazon Linux AMI, as well as the Canonical Ubuntu AMIs. For more details on these approaches, see the Running Commands on Your Linux Instance at Launch article.

Game servers

There are some game play scenarios that work well with an event driven RESTful model, for example, turn-based play and appointment games which don't require constant real-time updates can be built as stateless game servers with the techniques in the previous section.

Sometimes, however, a game server's approach needs to be the opposite of a RESTful approach. Clients establish a stateful two-way connection to the game server, via UDP, TCP, or WebSockets, enabling both the client and server to initiate messages. If the network connection is interrupted, the client must perform reconnect logic, and possibly logic to reset its state as well. Stateful game servers introduce challenges for automatic scaling because clients can't simply be round robin load balanced across a pool of servers.

Historically, many games used stateful connections and long-running server processes for all of their game functionality, especially in the case of larger AAA and MMO games. If you have a game that is architected in this manner, you can run it on AWS. We offer a managed service in Amazon GameLift that aids you in deploying, operating, and scaling dedicated game servers for session-based multiplayer games. You can also choose to run your own orchestration for game servers that uses Amazon EC2. Both are good choices depending on your requirements. However, for new games, we encourage you to use HTTP as much as possible, and only use stateful sockets for aspects of your game that really need it (such as online multiplayer).

The following table lists several packages that allow you to build event-driven servers.

Table 4: Packages to build event-driven servers

Language	Package
Node.js	Core, socket.io, Async
Python	Gevent, Twisted
Java	JBoss, Netty
Go	Socket.io
Erlang	Core
Ruby	Event Machine

C++ isn't listed in the table because it tends to be the language of choice for multiplayer game servers. Many commercial game engines, such as Amazon Lumberyard and Unreal Engine, are written in C++. This enables you to take existing game code from the client and reuse it on the server. This is particularly valuable when running physics or other frameworks on the server (such as Havok), which frequently only support C++. However, though there are packages that allow building event driven services, they tend to be more complex than those in the above list. Also, you wouldn't typically be running game simulation code in an event based service.

Regardless of programming language, stateful socket servers generally benefit from as large an instance as possible, since they are more sensitive to issues such as network latency. The largest instances in the Amazon EC2 compute-optimized instance family (for example, c5.*) are often the best options. These new-generation instances use enhanced networking via single root I/O virtualization (SR-IOV), which provides high packets per second, lower latency, and low jitter. This makes them ideal for game servers.

Matchmaking

Matchmaking is the feature that gets players into games. Typically, matchmaking follows a process like the following:

1. Ask the user about the type of game they would like to join (for example, deathmatch, time challenge, etc.).
2. Look at what game modes are currently being played online.
3. Factor in variables such as the user's geolocation (for latency) or ping time, language, and overall ranking.
4. Place the user on a game server that contains a matching game.

Games servers require long-lived processes, and they can't simply be round-robin load balanced in the way that you can with an HTTP request. After a player is on a given server, they remain on that server until the game is over, which could be minutes or hours.

In a modern cloud architecture, you should minimize your usage of long-running game server processes to only those gameplay elements that require it. For example, imagine an MMO or open-world shooter game. Some of the functionality, such as running around the world and interacting with other players, requires long-running game server processes. However, the rest of the API operations, such as listing friends, altering inventory, updating stats, and finding games to play, can easily be mapped to a REST web API.

In this approach, game clients would first connect to your REST API, and request a stateful game server. Your REST API would then perform matchmaking logic, and give clients an IP address and port of a server to connect to. The game client then connects directly to that game server's IP address.

This hybrid approach gives you the best performance for your socket servers because clients can directly connect to the EC2 instances. At the same time, you still get the benefits of using HTTP-based calls for your main entry point.

For most matchmaking needs, Amazon GameLift provides a matchmaking system called FlexMatch. You would control FlexMatch via your REST API, making calls to the Amazon GameLift API to initiate matching and return results. You can find more information on FlexMatch in the Amazon GameLift Developer Guide. If FlexMatch doesn't suit you needs for matchmaking, you can find more information about implementing matchmaking in a custom serverless environment in Fitting the Pattern: Serverless Custom Matchmaking with Amazon GameLift on the AWS Game Tech Blog.

Routing messages with Amazon SNS

There are two main categories of messages in gaming: messages targeted at a specific user, like private chat or trade requests, and group messages, such as chat or gameplay packets. A common strategy for sending and receiving messages is to use a socket server with a stateful connection. If your player base is small enough so that everyone can connect to a single server, you can route messages between players simply by selecting different sockets. In most cases, though, you need to have multiple servers, which means those servers also need some way to route messages between themselves.

Routing messages between EC2 server instances is one use case where Amazon SNS can help. Let's assume you had player 1 on server A, who wants to send a message to player 2 on server C, as shown in the following figure. In this scenario, server A could look at locally connected players, and when it can't find player 2, server A can forward the message to an SNS topic, which then propagates the message to other servers.

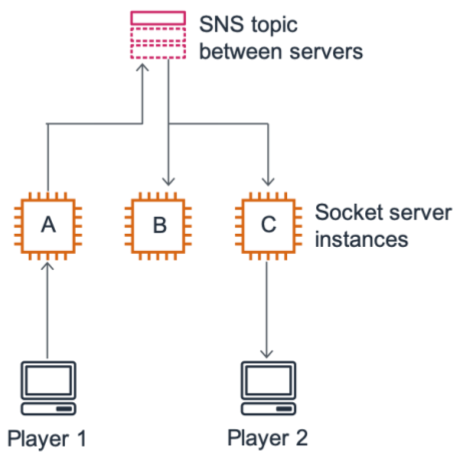


Figure 3: SNS-backed player to player communication between two servers

Amazon SNS fills a role here that is similar to a message queue such as RabbitMQ or Apache ActiveMQ. Instead of Amazon SNS, you could run RabbitMQ, Apache ActiveMQ, or a similar package on Amazon EC2. The advantage of Amazon SNS is that you don't have to spend time administering and maintaining queue servers and software on your own. For more information about Amazon SNS, see What is Amazon Simple Notification Service? and Create a Topic in the Amazon SNS Developer Guide.

Mobile push notifications

Unlike the previous use case, which is designed to handle near real-time in-game messaging, mobile push is best choice for sending a user a message when they are out of game, to draw them back in. An example might be a user-specific event, such as a friend beating your high score, or a broader game event such as a Double-XP Weekend.

Although Amazon SNS supports the ability to send push notifications directly to mobile clients, a better choice would be Amazon Pinpoint which provides not just mobile push notifications, but also e-mail, voice messages and SNS messaging, allowing a player pleasing multiple channel notification solution.

Last thoughts on game servers

It's easy to become obsessed with finding the perfect programming framework or pattern. Both RESTful and stateful game servers have their place, and any of the languages discussed previously will work well if programmed thoughtfully. More importantly, you need to spend time thinking about your overall game data architecture—where data lives, how to query it, and how to efficiently update it.

Relational vs. NoSQL databases

The advent of horizontally scaled applications has changed the application tier, and the traditional approach of a single large relational database. A number of new databases have become popular that eschew traditional Atomicity, Consistency, Isolation, and Durability (ACID) concepts in favor of lightweight access, distributed storage, and eventual consistency. These NoSQL databases can be especially beneficial for games, where data structures tend to be lists and sets (for example, friends, levels, items) as opposed to complex relational data.

As a general rule, the biggest bottleneck for online games tends to be database performance. A typical web-based app has a high number of reads and few writes. Think of reading blogs, watching videos, and so forth. Games are quite the opposite, with reads and writes frequently hitting the database due to constant state changes in the game.

There are many database options out there for both relational and NoSQL flavors, but the ones used most frequently for games on AWS are Amazon Aurora, Amazon ElastiCache for Redis, Amazon DynamoDB, Amazon RDS for MySQL and Amazon DocumentDB (with MongoDB compatibility.)

First, we'll cover MySQL because it's applicable to gaming and remains very popular. Combinations such as MySQL and Redis, or MySQL and DynamoDB, are very successful on AWS. All of the database alternatives described in this section support atomic operations such as increment and decrement, which are crucial for gaming.

MySQL

As an ACID-compliant relational database, MySQL has the following advantages:

- Transactions – MySQL provides support for grouping multiple changes into a single atomic transaction that must be committed or rolled back. NoSQL stores typically lack multi-step transactional functionality.
- Advanced querying – Since MySQL speaks SQL, this provides the flexibility to perform complex queries that evolve over time. NoSQL databases typically only support access by key or a single secondary index. This means you must make careful data design decisions up front.

- Single source of truth – MySQL guarantees data consistency internally. Part of what makes many NoSQL solutions faster is distributed storage and eventual consistency. (Eventual consistency means you could write a key on one node, fetch that key on another node, and have it not be there immediately.)
- Extensive tools – MySQL has extensive debugging and data analysis tools available for it. In addition, SQL is a general-purpose language that is widely understood.

These advantages continue to make MySQL attractive, especially for aspects of gaming such as account records, in-app purchases, and similar functionality where transactions and data consistency are paramount. Even gaming companies that are leveraging NoSQL offerings such as Redis and DynamoDB frequently continue to put transactional data such as accounts and purchases in MySQL.

If you're using MySQL on AWS, we recommend that you use Amazon RDS to host MySQL because it can save you valuable deployment and support cycles. Amazon RDS for MySQL automates the time-consuming aspects of database management such as launching EC2 instances, configuring MySQL, attaching Amazon Elastic Block Store (Amazon EBS) volumes, setting up replication, running nightly backups, and so on. In addition, Amazon RDS offers advanced features including synchronous Multi-AZ replication for high availability, automated primary/replica failover, and read replicas for increased performance. To get started with Amazon RDS, see Getting Started with Amazon RDS.

The following table includes some configuration options that we recommend you implement when you create your RDS MySQL DB instances.

Table 5: Recommended settings per environment

Option	Development/Test	Production
DB instance class	Micro	Medium or larger
Multi-AZ deployment	No	Yes (enables synchronous Multi-AZ replication and failover). For best performance, always launch production on an RDS DB instance that is separate from any of your Amazon RDS development/test DB instances.

Option	Development/Test	Production
Auto Minor Version Upgrade	Yes	Yes
Allocated Storage	5 GB	100 GB minimum (to enable Provisioned IOPS)
Use Provisioned IOPS	N/A	Yes Provisioned IOPS guarantees you a certain level of disk performance, which is important for large write loads. For more information about PIOPS, see Amazon RDS Provisioned IOPS Storage to Improve Performance.

Consider these additional guidelines when you create your RDS MySQL DB instances:

- Schedule Amazon RDS backup snapshots and upgrades during your low player count times, such as early morning. If possible, avoid running background jobs or nightly reports during this window, to prevent a query backlog.
- To find and analyze slow SQL queries in production, ensure you have enabled the MySQL slow query log in Amazon RDS as shown in the following list. These settings are configured using Amazon RDS DB Parameter Groups. Note that there is a minor performance penalty for the slow query log.
  - Set `slow_query_log` = 1 to enable. In Amazon RDS, slow queries are written to the `mysql.slow_log` table.
  - The value set in `long_query_time` determines that only queries that take longer than the specified number of seconds are included. The default is 10. Consider decreasing this value to 5, 3, or even 1.
  - Make sure to periodically rotate the slow query log as described in Common DBA Tasks for MySQL DB Instances in the Amazon RDS User Guide.

As your game grows and your write load increases, resize your RDS DB instances to scale up. Resizing an RDS DB instance requires some downtime, but if you deploy it in Multi-AZ mode as you would for production, this is limited to the time it takes to initiate a failover (typically a few minutes). For more information, see Modifying a DB Instance Running the MySQL Database Engine in the Amazon RDS User Guide. In addition, you can add one or more Amazon RDS read replicas to offload reads from your master RDS instance, leaving more cycles for database writes. For instructions on deploying replicas with Amazon RDS, see Working with Read Replicas.

Amazon Aurora

Amazon Aurora is a MySQL-compatible relational database engine that combines the speed and availability of high-end commercial databases with the simplicity and cost-effectiveness of open source databases. There are several key features that Amazon Aurora brings to a gaming workload:

- High performance – Amazon Aurora is designed to provide up to 5x the throughput of standard MySQL running on the same hardware. This performance is on par with commercial databases, for a significantly lower cost. On the largest Amazon Aurora instances, it’s possible to provide up to 500,000 reads and 100,000 writes per second, with 10 millisecond latency between read replicas.
- Data durability – In Amazon Aurora, each 10 GB chunk of your database volume is replicated six ways across three Availability Zones, allowing for the loss of two copies of data without affecting database write availability, and three copies without affecting read availability. Backups are done automatically and continuously to Amazon S3, which is designed for 99.999999999% durability with a retention period of up to 35 days. You can restore your database to any second during the retention period, up to the last five minutes.
- Scalability – Amazon Aurora is capable of automatically scaling its storage subsystem out to 64 TB of storage. This storage is automatically provisioned for you so that you don’t have to provision storage ahead of time. As an added benefit, this means you pay only for what you use, reducing the costs of scaling. Amazon Aurora also can deploy up to 15 read replicas in any combination of Availability Zones, including cross-Region where Amazon Aurora is available. This allows for seamless failover in case of an instance failure.

The following are some recommendations for using Amazon Aurora in your gaming workload:

- Use the following DB instance classes: t2.small instance in you development/test environments and r3.large or larger instance in you production environment.
- Deploy read replicas in at least one additional Availability Zone to provide for failover and read operation offloading.

- Schedule Amazon RDS backup snapshots and upgrades during low player count times. If possible, avoid running jobs or reports against the database during this window to prevent backlogging.

If your game grows beyond the bounds of a traditional relational database like MySQL or Amazon Aurora, we recommend that you perform a performance evaluation, including tuning parameters and sharding. In addition, you should look at using a NoSQL offering, such as Redis or DynamoDB, to offload some workloads from MySQL. In the following sections, we'll cover a few popular NoSQL offerings.

Redis

Best described as an atomic data structure server, Redis has some unique features not found in other databases. Redis provides foundational data types such as counters, lists, sets, and hashes, which are accessed using a high-speed text-based protocol. For details on available Redis data types, see the Redis data type documentation and An introduction to Redis data types and abstractions. These unique data types make Redis an ideal choice for leaderboards, game lists, player counts, stats, inventories, and similar data. Redis keeps its entire data set in memory so access is extremely fast. For comparisons with Memcached, see Redis Benchmarks.

There are a few caveats concerning Redis that you should be aware of. First, you need a large amount of physical memory because the entire dataset is memoryresident (that is, there is no virtual memory support). Replication support is also simplistic, and debugging tools for Redis are limited. Redis is not suitable as your only data store. But when used in conjunction with a disk-backed database, such as MySQL or DynamoDB, Redis can provide a highly scalable solution for game data. Redis plus MySQL is a very popular solution for gaming.

Redis uses minimal CPU, but it uses lots of memory. As a result, it's best suited to high-memory instances, such as the Amazon EC2 memory-optimized instance family (that is, r3.*). AWS offers a fully-managed Redis service, Amazon ElastiCache for Redis. ElastiCache for Redis can handle clustering, primary/replica replication, backups, and many other common Redis maintenance tasks. For a deep dive on getting the most out of ElastiCache, see the AWS whitepaper Performance at Scale with Amazon ElastiCache.

MongoDB

MongoDB is a document-oriented database, which means that data is stored in a nested data structure, similar to a structure you would use in a typical programming language. MongoDB uses a binary variant of JSON called BSON for communication, which makes programming against it a matter of storing and retrieving JSON structures. This has made MongoDB popular for games and web applications, since server APIs are usually JSON, too.

MongoDB also offers a number of interesting hybrid features, including a SQL-like syntax that enables you to query data by range and composite conditions. MongoDB supports atomic operations such as increment/decrement and add/remove from list; this is similar to Redis support for these operations. For examples of atomic operations that MongoDB supports, see the MongoDB documentation on findAndModify.

MongoDB is widely used as a primary data store for games, and is frequently used in conjunction with Redis, since the two complement each other well. Transient game data, sessions, leaderboards, and counters are kept in Redis, and then progress is saved to MongoDB at logical points (for example, at the end of a level or when a new achievement is unlocked). Redis yields high-speed access for latency-sensitive game data, and MongoDB provides simplified persistence.

MongoDB supports native replication and sharding as well, although you do have to configure and monitor these features yourself. For an in-depth look at deploying MongoDB on AWS, see the AWS whitepaper MongoDB on AWS.

Amazon DocumentDB (with MongoDB compatibility) is a fully managed document database service that supports MongoDB workloads. It's designed for high availability, performance at scale, and is highly secure.

Amazon DynamoDB

Finally, DynamoDB is a fully managed NoSQL solution provided by AWS. DynamoDB manages tasks such as synchronous replication and IO provisioning for you, in addition to automatic scaling and managed caching. DynamoDB uses a Provisioned Throughput model, where you specify how many reads and writes you want per second, and the rest is handled for you under the hood.

To set up DynamoDB, see the Getting Started Guide. Games frequently use DynamoDB features in the following ways:

- Key-value store for user data, items, friends, and history.
- Range key store for leaderboards, scores, and date-ordered data.
- Atomic counters for game status, user counts, and matchmaking.

Like MongoDB and MySQL, DynamoDB can be paired with a technology such as Redis to handle real-time sorting and atomic operations. Many game developers find DynamoDB to be sufficient on its own, but the point is you still have the flexibility to add Redis or a caching layer to a DynamoDB-based architecture. Let's revisit our reference diagram with DynamoDB to see how it simplifies the architecture.

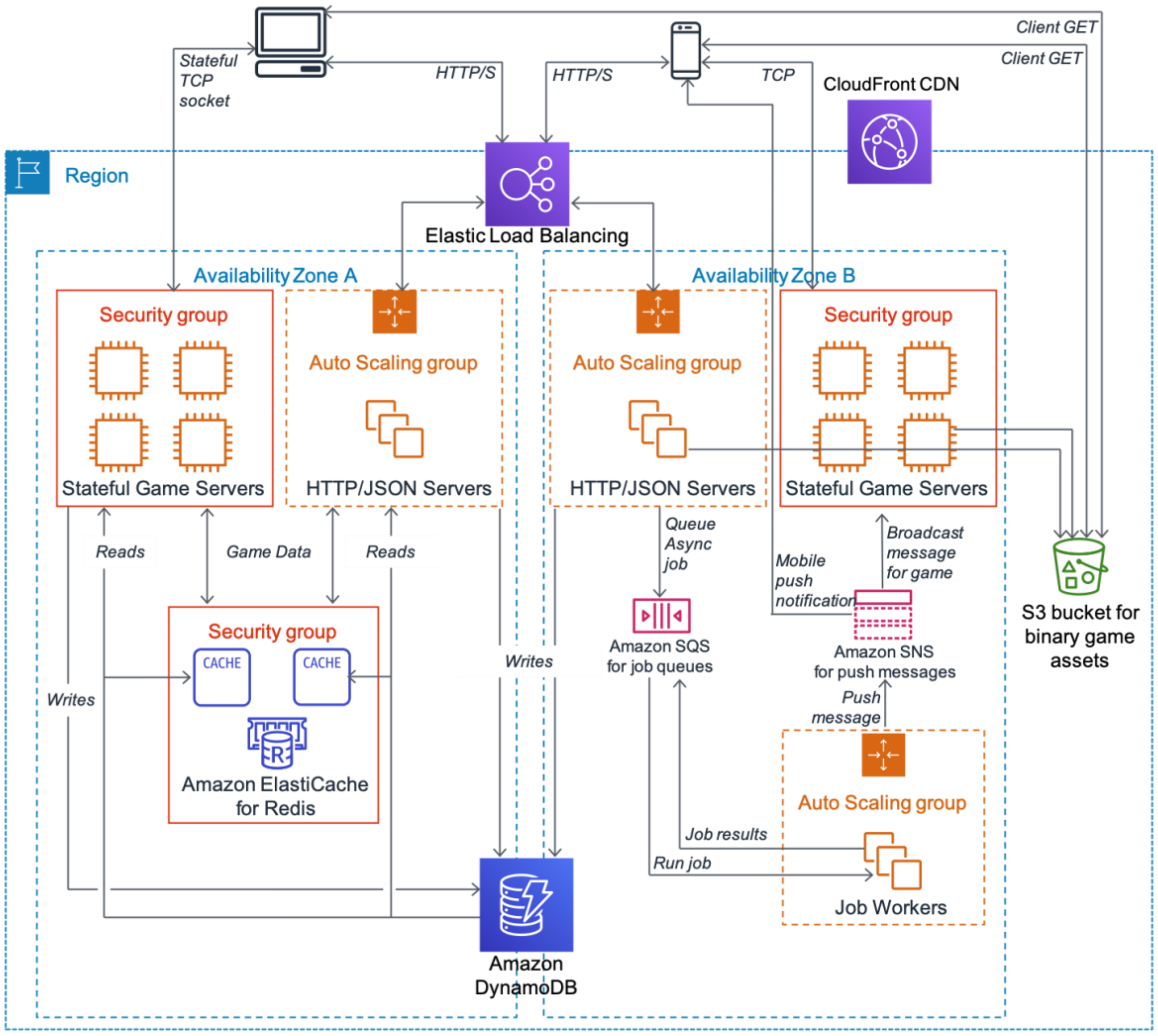


Figure 4: A fully production-read game backend running on AWS using DynamoDB

Table structure and queries

DynamoDB, like MongoDB, is a loosely structured NoSQL data store that allows you to save different sets of attributes on a per-record basis. You only need to predefine the primary key strategy you're going to use:

- Partition key – The partition key is a single attribute that DynamoDB uses as input to an internal hash function. This could be a player name, game ID, UUID, or similar unique key. Amazon DynamoDB builds an unordered hash index on this key.
- Partition key and sort key – Referred to as a composite primary key, this type of key is composed of two attributes: the partition key and the sort key. DynamoDB uses the partition key value as input to an internal hash function, and all items with the same partition key are stored together, in sorted order by sort key value. For example, you could store game history as a duplet of [user_id, last_login]. Amazon DynamoDB builds an unordered hash index on the partition key attribute, and a sorted range index on the sort key attribute. Only the combination of both keys is unique in this scenario.

For best querying performance, you should maintain each DynamoDB table at a manageable size. For example, if you have multiple game modes, it's better to have a separate leaderboard table for each game mode, rather than a single giant table. This also gives you the flexibility to scale your leaderboards separately, in the event that one game mode is more popular than the others.

Provisioned throughput

DynamoDB shards your data behind the scenes to give you the throughput you've requested. DynamoDB uses the concept of read and write units. One read capacity unit represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. One write capacity unit represents one write per second for an item up to 1 KB in size. The defaults are 5 read and 5 write units, which means 20 KB of strongly consistent reads/second and 5 KB of writes/second. You can increase your read and or write capacity at any time by any amount up to your account limits. You can also decrease the read and or write capacity by any amount but this can't exceed more than four decreases in one day. Scaling can be done using the AWS Management Console or AWS CLI by selecting the table and modifying it appropriately. You can also take advantage of DynamoDB Auto Scaling by using the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. DynamoDB Auto Scaling works in conjunction with Amazon CloudWatch alarms that monitor the capacity units. It scales according to your defined rules.

There is a delay before the new provisioned throughput is available, while data is repartitioned in the background. This doesn't cause downtime, but it does mean that the DynamoDB scaling is best suited for changes over time, such as the growth of a game from 1,000 to 10,000 users. It isn't designed to handle hourly user spikes. For this, as with other databases, you need to leverage some form of caching to add resiliency.

To get the best performance from DynamoDB, make sure your reads and writes are spread as evenly as possible across your keys. Using a hexadecimal string such as a hash key or checksum is one easy strategy to inject randomness. For more details on optimizing DynamoDB performance, see Best Practices for DynamoDB in the Amazon DynamoDB Developer Guide.

Amazon DynamoDB Accelerator (DAX)

DAX allows you to provision a fully-managed, in-memory cache for DynamoDB that speeds up the responsiveness of your DynamoDB tables from millisecond-scale latency to microseconds. This acceleration comes without requiring any major changes in your game code, which simplifies deployment into your architecture. All you have to do is re-initialize your DynamoDB client with a new endpoint that points to DAX and the rest of the code can remain untouched. DAX handles cache invalidation and data population without your intervention. This cache can help speed responsiveness when running events that might cause a spike in players, such as a seasonal DLC offering, or a new patch release.

Other NoSQL options

There are a number of other NoSQL alternatives, including Riak, Couchbase, and Cassandra. You can use any of these for gaming, and there are examples of gaming companies using them on AWS with success. As with choosing a server programming language, there is no perfect database—you need to weigh the pros and cons of each one.

Caching

For gaming, adding a caching layer in front of your database for frequently used data can alleviate a significant number of scalability problems. Even a short-lived cache of just a few seconds for data such as leaderboards, friend lists, and recent activity can greatly offload your database. Adding cache servers is also cheaper than adding additional database servers, so it also lowers your AWS costs

Memcached is a high-speed, memory-based key-value store that is the gold standard for caching. Redis features similar performance to Memcached, plus Redis has advanced data types. Both options perform well on AWS. You can choose to install Memcached or Redis on EC2 instances yourself, or you can use Amazon ElastiCache, the AWS managed caching service. Like Amazon RDS and DynamoDB, ElastiCache completely automates the installation, configuration, and management of Memcached and Redis on AWS. For more details on setting up ElastiCache, see Getting Started with Amazon ElastiCache in the Amazon ElastiCache User Guide.

ElastiCache groups servers in a cluster to simplify management. Most ElastiCache operations like configuration, security, and parameter changes are performed at the cache cluster level. Despite the use of the cluster terminology, ElastiCache nodes do not talk to each other or share cache data. ElastiCache deploys the same versions of Memcache and Redis that you would download yourself, so existing client libraries written in Ruby, Java, PHP, Python, and so on are completely compatible with ElastiCache.

The typical approach to caching is known as lazy population or cache aside. This means that the cache is checked, and if the value is not in cache (a cache miss), the record is retrieved, stored in cache, and returned. The following Python example checks ElastiCache for a value, queries the database if the cache doesn’t have it, and then stores the value back to ElastiCache for subsequent queries.

Lazy population is the most prevalent caching strategy because it only populates the cache when a client actually requests the data. This way it avoids extraneous writes to the cache in the case of records that are infrequently (or never) accessed, or that change before being read. This pattern is so ubiquitous that most major web development frameworks such as Rails, Django, and Grails include plugins that wrap this strategy. The downside to this strategy is that when data changes, the next client that requests it incurs a cache miss, which means that their response time is slower because the new record needs to be queried from the database and populated into cache.

This downside leads us to the second most prevalent caching strategy. For data that you know will be accessed frequently, populate the cache when records are saved to avoid unnecessary cache misses. This means that client response times will be faster and more uniform. In this case, you simply populate the cache when you update the record, rather than when the next client queries it. The tradeoff here is that it could result in an unnecessarily high number of cache writes if your data is changing rapidly. In addition, writes to the database can appear slower to users, since the cache also needs to be updated.

To choose between these two strategies you need to know how often your data is changing versus how often it's being queried.

The final popular caching alternative is a timed refresh. This is beneficial for data feeds that span multiple different records, such as leaderboards or friend lists. In this strategy, you would have a background job that queries the database and refreshes the cache every few minutes. This decreases the write load on your cache, and enables additional caching to happen upstream (for example, at the CDN layer) because pages remain stable longer.

Amazon ElastiCache scaling

ElastiCache simplifies the process of scaling your cache instances up and down. ElastiCache provides access to a number of Memcached metrics in CloudWatch at no additional charge. You should set CloudWatch alarms based on these metrics to alert you to cache performance issues. You can configure these alarms to send emails when the cache memory is almost full, or when cache nodes are taking a long time to respond. We recommend that you monitor the following metrics:

- CPUUtilization – How much CPU Memcached or Redis is using. Very high CPU could indicate an issue.
- Evictions – Number of keys that have to be forced out of memory due to lack of space. Should be zero. If it's not near zero, you need a larger ElastiCache instance.
- GetHits/CacheHits and GetMisses/CacheMisses – How frequently does your cache have the keys you need? The higher percentage of hits, the more you're offloading your database.
- CurrConnections – The number of clients that are currently connected (this depends on the application).

In general, monitoring hits, misses, and evictions is sufficient for most applications. If the ratio of hits to misses is too low, you should revisit your application code to make sure your cache code is working as expected. As mentioned, typically evictions should be zero 100 percent of the time. If evictions are nonzero, either scale up your ElastiCache nodes to provide more memory capacity or revisit your caching strategy to ensure you're only caching what you need to cache.

Additionally, you can configure your cache node cluster to span multiple Availability Zones to provide high availability for your game's caching layer. This ensures that in the event of an Availability Zone being unavailable, your database is not overwhelmed by a sudden spike in requests. When creating a cache cluster or adding nodes to an existing cluster, you can chose the Availability Zones for the new nodes. You can either specify the requested number of nodes in each Availability Zone or select the option to spread nodes across zones.

With Amazon ElastiCache for Redis you can create a read replica in another Availability Zone. Upon a failure of the primary node, AWS provisions a new primary node. In scenarios where the primary node cannot be provisioned, you can decide which read replica to promote to be the new primary.

ElastiCache for Redis also supports Sharded Cluster with supported Redis Engines version 3 or higher. You can create clusters with up to 15 shards, expanding the overall in-memory data store to more than 3.5 TiB. Each shard can have up to 5 read replicas, giving you the ability to handle 20 million reads and 4.5 million writes per second.

The sharded model, in conjunction with the read replicas, improves overall performance and availability. Data is spread across multiple nodes and the read replicas support rapid, automatic failover in the event that a primary node has an issue.

To take advantage of the sharded model, you must use a Redis client that is cluster-aware. The client will treat the cluster as a hash table with 16,384 slots spread equally across the shards, and will then map the incoming keys to the proper shard. ElastiCache for Redis treats the entire cluster as a unit for backup and restore purposes. You don't have to think about or manage backups for the individual shards

Binary game content with Amazon S3

Your database is responsible for storing user data, including accounts, stats, items, purchases, and so forth. But for game-related binary data, Amazon S3 is a better fit. Amazon S3 provides a simple HTTP-based API to `PUT` (upload) and `GET` (download) files. With Amazon S3, you pay only for the amount of data that you store and transfer. Using Amazon S3 consists of creating a bucket to store your data in, and then making HTTP requests to and from that bucket. For a walkthrough of the process, see Create a Bucket in the Amazon S3 Getting Started Guide.

Amazon S3 is ideally suited for a variety of gaming use cases, including the following:

- Content downloads – Game assets, maps, patches, and betas
- User-generated files – Photos, avatars, user-created levels, and device backups
- Analytics – Storing metrics, device logs, and usage patterns

- Cloud saves – Game save data and syncing between devices (AWS AppSync would be a good choice as well)

Although you can store this type of data in a database, using Amazon S3 has a number of advantages, including the following:

- Storing binary data in a DB is memory and disk intensive, consuming valuable query resources.
- Clients can directly download the content from Amazon S3 using a simple HTTP/S GET.
- Designed for 99.999999999% durability and 99.99% availability of objects over a given year
- Amazon S3 natively supports features such as ETag, authentication, and signed URLs.
- Amazon S3 plugs into the Amazon CloudFront CDN for distributing content quickly to large numbers of clients.

With these factors in mind, let's look at the aspects of Amazon S3 that are most relevant for gaming.

Content delivery and Amazon CloudFront

Downloadable content (DLC) is a huge aspect of modern games from an engagement perspective, and it is becoming a primary revenue stream. Users expect an ongoing stream of new characters, levels, and challenges for months—if not years—after a game's release. Being able to deliver this content quickly and cost-effectively has a big impact on the profitability of a DLC strategy.

Although the game client itself is typically distributed through a given platform's app store, pushing a new version of the game just to make a new level available can be onerous and time consuming. Promotional or time-limited content, such as Halloween-themed assets or a long weekend tournament, are usually easier to manage yourself, in a workflow that mirrors the rest of your server infrastructure.

If you're distributing content to a large number of clients (for example, a game patch, expansion, or beta), we recommend that you use Amazon CloudFront in front of Amazon S3. CloudFront has points of presence (POPs) located throughout the world, which improves download performance. In addition, you can configure which Regions

CloudFront serves to optimize your costs. For more information, see the CloudFront FAQ, in particular *How does CloudFront lower my costs?*

Finally, if you anticipate significant CloudFront usage, you should contact our CloudFront sales team because Amazon offers reduced pricing that is even lower than our on-demand pricing for high-usage customers.

Easy versioning with ETag

As mentioned earlier, Amazon S3 supports HTTP ETag and the If-None-Match HTTP header, which are well known to web developers but frequently overlooked by game developers. These headers enable you to send a request for a piece of Amazon S3 content, and include the MD5 checksum of the version you already have. If you already have the latest version, Amazon S3 responds with an HTTP 304 Not Modified, or HTTP 200 along with the file data, if you need it.

Leveraging ETag in this manner makes any future use of CloudFront more powerful because CloudFront also supports the Amazon S3 ETag. For more information, see Request and Response Behavior for Amazon S3 Origins in the Amazon CloudFront Developer Guide.

Finally, you also have the ability to Geo Target or Restrict access to your content through CloudFront's Geo Targeting feature. Amazon CloudFront detects the country where your customers are located and will forward the country code to your origin servers. This allows your origin server to determine the type of personalized content that will be returned to the customer based on their geographic location. This content could be anything from a localized dialog file for an RPG to localized asset packs for your game.

Uploading content to Amazon S3

Our other gaming use cases for Amazon S3 revolve around uploading data from the game, be it user-generated content, analytics, or game saves. There are two strategies for uploading to Amazon S3: either upload directly to Amazon S3 from the game client, or upload by first posting to your REST API servers, and then have your REST servers upload to Amazon S3. Although both methods work, we recommend uploading directly to Amazon S3 if possible, since this offloads work from your REST API tier.

Uploading directly to Amazon S3 is straightforward, and can even be accomplished directly from a web browser. For more information, see Browser-Based Uploads Using POST (AWS Signature Version 2) in the Amazon S3 Developer Guide. You can even create secure URLs for players to upload content (such as from an out of game tool) using pre-signed URLs.

To protect against corruption, you should also consider calculating an MD5 checksum of the file and including it in the Content-MD5 header. This approach enables Amazon S3 to automatically verify the file was not corrupted during upload. For more information, see PUT Object in the Amazon S3 API Reference.

User-generated content (UGC) is a great use case for uploading data to Amazon S3. A typical piece of UGC has two parts: binary content (for example, a graphic asset), and its metadata (for example, name, date, author, tags, etc.). The usual pattern is to store the binary asset in Amazon S3, and then store the metadata in a database. Then, you can use the database as your master index of available UGC that others can download.

The following figure shows an example call flow that you can use to upload UGC to Amazon S3.

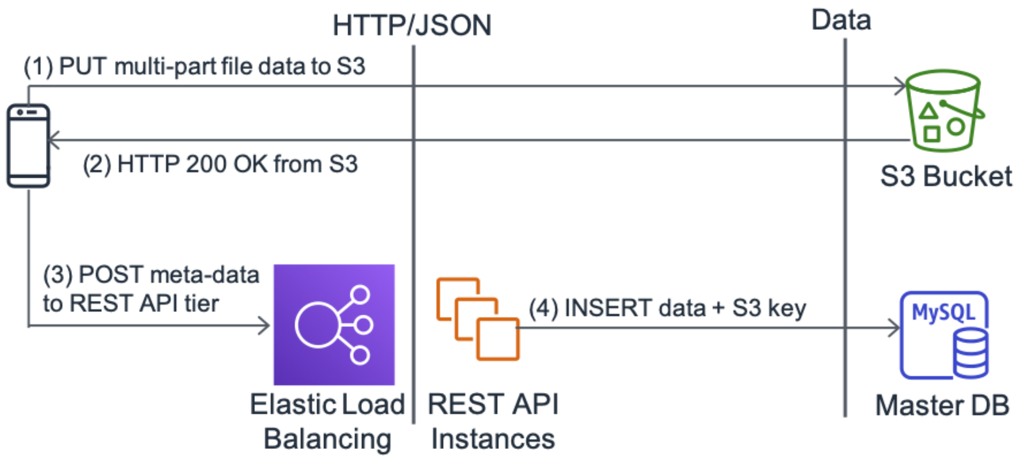


Figure 5: A simple workflow for transfer of game content

In this example, first you PUT the binary game asset (for example, avatar, level, etc.) to Amazon S3, which creates a new object in Amazon S3. After you receive a success response from Amazon S3, you make a POST request to our REST API layer with the metadata for that asset. The REST API needs to have a service that accepts the Amazon S3 key name, plus any metadata you want to keep, and then it stores the key name and the metadata in the database. The game's other REST services can then query the database to find new content, popular downloads, and so on.

This simple call flow handles the case where the asset data is stored verbatim in Amazon S3, which is usually true of user-generated levels or characters. This same pattern works for game saves as well—store the game save data in Amazon S3, and then index it in your database by user_id, date, and any other important metadata. If you need to do additional processing of an Amazon S3 upload (for example, generating preview thumbnails), make sure to read the section on Asynchronous Jobs later in this book. In that section, we'll discuss adding Amazon SQS to queue jobs to handle these types of tasks.

Analytics and A/B testing

Collecting data about your game is one of the most important things you can do, and one of the easiest as well. Perhaps the trickiest part is deciding what to collect. You should consider keeping track of any reasonable metrics you can think of for a user (for example, total hours played, favorite characters or items, current and highest level, etc.) If you aren't sure what to measure, or if you have a client that is not updated easily. Amazon S3 is a popular choice for storing raw metrics data as it can be very cost effective.

However, if you are able to formulate questions that you want answered beforehand or if client updates are simple to distribute, you can focus on gathering the data that help you answer those specific questions.

After you've identified the data, follow these steps to track it:

1. Collect metrics in a local data file on the user's device (for example, mobile, console, PC, etc.). To make things easier later, we recommend using a CSV format and a unique filename. For example, a given user might have their data tracked in 241-game_name-user_idYYYYMMDDHHMMSS.csv or something similar.
2. Periodically persist the data by having the client upload the metrics file directly to Amazon S3. Or, you can integrate with Amazon Kinesis and adopt a loosely coupled architecture, as we discussed previously. When you go to upload a given data file to Amazon S3, open a new local file with a new file name. This keeps the upload loop simple.

3. For each file you upload, put a record somewhere indicating that there's a new file to process. Amazon S3 event notifications provide an excellent way to support this. To enable notifications, you must first add a notification configuration identifying the events you want Amazon S3 to publish, such as a file upload, and the destinations where you want Amazon S3 to send the event notifications. We recommend Amazon SQS because you can then have a background worker listening to Amazon SQS for new files, and processing them as they arrive. For more details, see the Amazon SQS section in this book.
4. As part of a background job, process the data using a framework such as Amazon EMR, or other framework that you choose to run on Amazon EC2. This background process can look at new data files that have been uploaded since the last run, and perform aggregation or other operations on the data. (Note that if you're using Amazon EMR, you may not need step #3 because Amazon EMR has built-in support for streaming new files.)
5. Optionally, feed the data into Amazon Redshift for additional data warehousing and analytics flexibility. Amazon Redshift is an ANSI SQL-compliant, columnar data warehouse that you pay for by the hour. This enables you to perform queries across large volumes of data, such as sums and min/max using familiar SQL-compliant tools.

Repeat these steps in a loop, uploading and processing data asynchronously.

The following figure shows how this pattern works.

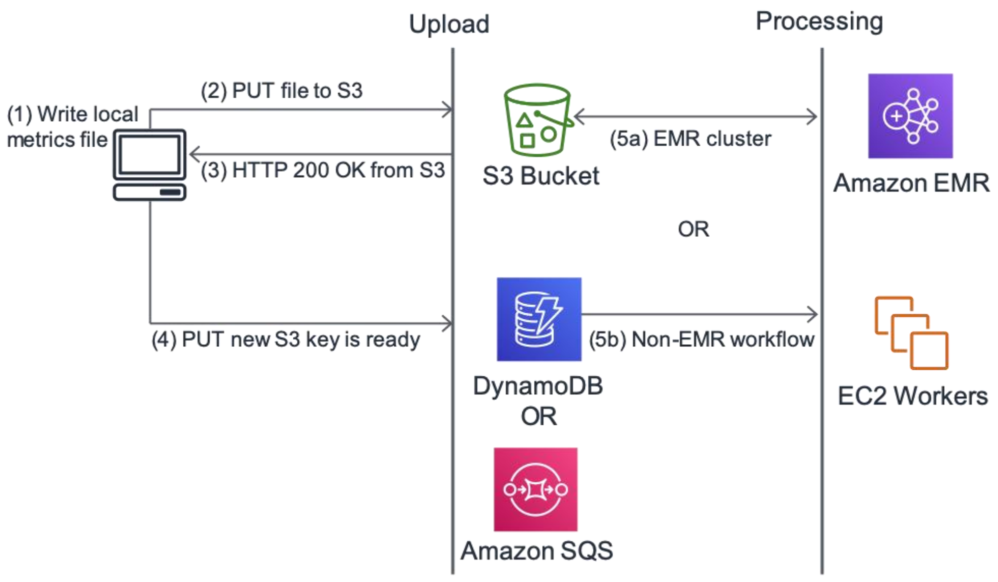


Figure 6: A simple pipeline for analytics and A/B testing

For both analytics and A/B testing, the data flow tends to be unidirectional. That is, metrics flow in from users, are processed, and then a human makes decisions that affect future content releases or game features. Using A/B testing as an example, when you present users with different items, screens, and so forth, you can make a record of the choice they were given, along with their subsequent actions (for example, purchase, cancel, etc.). Then, periodically upload this data to Amazon S3, and use Amazon EMR to create reports. In the simplest use case, you could just generate cleaned up data from Amazon EMR in CSV format into another Amazon S3 bucket, and then load this into a spreadsheet program.

For more information on analytics and Amazon EMR, see Data Lakes and Analytics on AWS and the Amazon EMR Documentation.

Amazon Athena

Gleaning insights quickly and cheaply is one of the best ways that developers can improve on their games. Traditionally, this has been relatively difficult because data normally has to be extracted from game application servers, stored somewhere, transformed, and then loaded into a database in order to be queried later. This process can take a significant amount of time and compute resources, greatly increasing the cost of running such tasks.

Amazon Athena assists with your analytical pipeline by providing the means of querying data stored in Amazon S3 using standard SQL. Because Athena is serverless, there is no infrastructure to provision or manage, and generally there is no requirement to transform data before applying a schema to start querying. However, keep the following points in mind to optimize performance while using Athena for your queries:

- **Ad hoc queries** – Because Athena is priced at a base of $5 per TB of data scanned, this means that you incur no charges when there aren’t any queries being run. Athena is ideally suited for running queries on an ad hoc basis when information must be gleaned from data quickly without running an extract, transform, and load (ETL) process first.
- **Proper partitioning** – Partitioning data divides tables into parts that keep related entries together. Partitions act as virtual columns. You define them at table creation, and they can help reduce the amount of data scanned per query, thereby improving performance and reducing the cost of any particular query. You can restrict the amount of data scanned by a query by specifying filters based on the partition.

For example, in the following query:

```
SELECT count(*) FROM lineitem WHERE l_gamedate = '2019-10-31'
```

A non-partitioned table would have to scan the entire table, looking through potentially millions of records, and gigabytes of data, slowing down the query, and adding unnecessary cost. A properly partitioned table can help speed queries, and significantly reduce cost by cutting the amount of data queried by Athena. For a detailed example, see Top 10 Performance Tuning Tips for Amazon Athena on the AWS Big Data Blog.

- **Compression** – Just like partitioning, proper compression of data can help reduce network load and costs by reducing data size. It's also best to make sure that the compression algorithm you choose allows for splittable files so Athena's execution engine can increase parallelism for additional performance.
- **Presto knowledge** –Presto is an open source distributed SQL query engine for running interactive analytic queries against data sources of all sizes ranging from gigabytes to petabytes. Athena uses Presto, and therefore understanding Presto can help you to optimize the various queries that you run on Athena. For example, the ORDER BY clause returns the results of a query in sort order. To do the sort, Presto must send all rows of data to a single worker and then sort them. This could cause memory pressure on Presto, which could cause the query to take a long time to execute. Worse, the query could fail. If you are using the ORDER BY clause to look at the top or bottom N values, then use a LIMIT clause to reduce the cost of the sort significantly by pushing the sorting and limiting to individual workers, rather than the sorting being done in a single worker.

Amazon S3 performance considerations

Amazon S3 can scale to tens of thousands of PUTs and GETs per second. To achieve this scale, there are a few guidelines you must follow to get the best performance out of Amazon S3. First, as with DynamoDB, make sure that your Amazon S3 key names are evenly distributed because Amazon S3 determines how to partition data internally based on the first few characters in the key name.

Let's assume your bucket is called `mygame-ugc` and you store files based on a sequential database ID:

```
http://mygame-ugc.s3.amazonaws.com/10752.dat

http://mygame-ugc.s3.amazonaws.com/10753.dat

http://mygame-ugc.s3.amazonaws.com/10754.dat

http://mygame-ugc.s3.amazonaws.com/10755.dat
```

In this case, all of these files would likely live in the same internal partition within Amazon S3 because the keys all start with 107. This limits your scalability, because it results in writes that are sequentially clustered together. A solution is to use a hash function to generate the first part of the object name, in order to randomize the distribution of names. One strategy is to use an MD5 or SHA1 of filename and prefix the Amazon S3 key with that, as shown in the following code example:

```
http://mygame-ugc.s3.amazonaws.com/988-10752.dat

http://mygame-ugc.s3.amazonaws.com/483-10753.dat

http://mygame-ugc.s3.amazonaws.com/d5d-10754.dat

http://mygame-ugc.s3.amazonaws.com/06f-10755.dat
```

Here's a variation with a Python SHA1 example:

```
#!/usr/bin/env python

import hashlib

sha1 = hashlib.sha1(filename).hexdigest()[0:3]

path = sha1 + "-" + filename
```

For more information about maximizing S3 performance, see Best Practices Design Patterns: Optimizing Amazon S3 Performance in the Amazon S3 Developer Guide. If you anticipate a particularly high PUT or GET load, file an AWS Support Ticket.

Loosely coupled architectures with asynchronous jobs

Loosely coupled architectures that involve decoupling components refers to the concept of designing your server components so that they can operate as independently as possible. A common approach is to put queues between services, so that a sudden burst of activity on one part of your system doesn't cascade to other parts. Some aspects of gaming are difficult to decouple, because data needs to be as up-to-date as possible to provide a good matchmaking and gameplay experience for players. However, most data, such as cosmetic or character data, doesn't have to be up-to-the-millisecond.

Leaderboards and avatars

Many gaming tasks can be decoupled and handled in the background. For example, the task of a user updating his stats must be done in real time, so that if a user exits and then re-enters the game, they won't lose progress. However, re-ranking the global top 100 leaderboard isn't required every time a user posts a new high score. Most users appear far down the leaderboard. Instead, the ranking process can be decoupled from score posting, and performed in the background every few minutes. This approach has minimal impact on the game experience because game ranks are highly volatile in any active online game.

As another example, consider allowing users to upload a custom avatar for their character. In this case, your front-end servers put a message into a queue such as Amazon SQS about the new avatar upload. You write a background job that runs periodically, pulls avatars off the queue, processes them, and marks them as available in MySQL, Aurora, DynamoDB, or whatever database you're using. The background job runs on a different set of EC2 instances, which can be set up to automatically scale just like your front-end servers. To help you get started quickly, Elastic Beanstalk provides worker environments that simplify this process by managing the Amazon SQS queue and running a daemon process on each instance that reads from the queue for you.

This approach is an effective way to decouple your front-end servers from backend processing, and it enables you to scale the two independently. For example, if the image resizing is taking too long, you can add additional job instances, without needing to scale your REST servers too.

The rest of this section focuses on Amazon SQS. Note that you could implement this pattern with an alternative such as RabbitMQ or Apache ActiveMQ deployed to Amazon EC2 instead.

Amazon SQS

Amazon SQS is a fully managed queue solution with a long-polling HTTP API. This makes it easy to interface with regardless of the server languages you're using. To get started with Amazon SQS, see Getting Started with Amazon SQS in the Amazon SQS Developer Guide.

Here are a few tips to best use Amazon SQS:

- Create your SQS queues in the same Region as your API servers to make writes as fast as possible. Your asynchronous job workers can live in any Region because they are not time-dependent. This enables you to run API servers in Regions near your users and job instances in more economical Regions.
- Amazon SQS is designed to scale horizontally. A given Amazon SQS client can process about 50 requests a second. The more Amazon SQS client processes you add, the more messages you can process concurrently. For tips on adding additional worker processes and EC2 instances, see Increasing Throughput with Horizontal Scaling and Action Batching in the Amazon SQS Developer Guide.
- Consider using Amazon EC2 Spot Instances for your job workers to save money. Amazon SQS is designed to redeliver messages that aren't explicitly deleted, which protects against EC2 instances going away mid-job. Make sure to delete messages only after you have completed processing them. This enables another EC2 instance to retry the job if a given instance fails while running.
- Consider message visibility, which you can think of this as the redelivery time if a message is not deleted. The default is 30 seconds. You may need to increase this if you have long-running jobs, to avoid multiple queue readers from receiving the same message.
- Amazon SQS also supports dead-letter queues. A dead-letter queue is a queue that other (source) queues can target for messages that can't be processed (consumed) successfully. You can set aside and isolate these messages in the dead-letter queue to determine why their processing doesn't succeed.

In addition, Amazon SQS has the following caveats:

- Messages are not guaranteed to arrive in order. You may receive messages in random order (for example, 2, 3, 5, 1, 7, 6, 4, 8). If you need strict ordering of messages, see the following FIFO Queues section.
- Messages typically arrive quickly, but occasionally a message may be delayed by a few minutes.
- Messages can be duplicated and it's the responsibility of the client to de-duplicate.

Taken together, this information means that you must make sure your asynchronous jobs are coded to be idempotent and resilient to delays. Resizing and replacing an avatar is a good example of idempotence because doing that twice would yield the same result.

Finally, if your job workload scales up and down over time (for example, perhaps more avatars are uploaded when more users are online), consider using Auto Scaling to Launch Spot Instances. Amazon SQS offers a number of metrics that you can automatically scale on, the best being ApproximateNumberOfMessagesVisible. The number of visible messages is basically your queue backlog. For example, depending on how many jobs you can process each minute, you could scale up if this reaches 100 and then scale back down when it falls below 10. For more information about Amazon SQS and Amazon SNS metrics, see Amazon SNS Metrics and Dimensions and Amazon SQS Metrics and Dimensions in the Amazon CloudWatch User Guide.

FIFO queues

Although the recommended method of using Amazon SQS is to engineer and architect for your application to be resilient to duplication and misordering, you may have certain tasks where the ordering of messages is absolutely critical to proper functioning and duplicates can’t be tolerated. For example, micro-transactions where a user wants to buy a particular item once, and only once, and this action must be strictly regulated.

To supplement this requirement, First-In-First-Out (FIFO) queues are available in select AWS Regions. FIFO queues provide the ability to process messages both in order, and exactly once. There are additional limitations when working with FIFO queues due to the emphasis on message order and delivery. For more details about FIFO queues, see FIFO (First-In-First-Out) Queues in the Amazon SQS Developer Guide.

Other queue options

In addition to Amazon SQS and Amazon SNS, there are dozens of other approaches to message queues that can run effectively on Amazon EC2, such as RabbitMQ, ActiveMQ, and Redis. With all of these, you are responsible for launching a set of EC2 instances and configuring them yourself, which is outside the scope of this book. Keep in mind that running a reliable queue is much like running a highly available database: you need to consider high-throughput disk (such as Amazon EBS PIOPS), snapshots, redundancy, replication, failover, and so forth. Ensuring the uptime and durability of a custom queue solution can be a time-consuming task, and can fail at the worst times like during your highest load peaks.

Cost of the cloud

With AWS you no longer need to dedicate valuable resources to building costly infrastructure, including purchasing servers and software licenses, or leasing facilities. With AWS you can replace large upfront expenses with lower variable costs and pay only for what you use and for as long as you need it. All AWS services are available on demand, and don't require long-term contracts or complex licensing dependencies. Some of the advantages of AWS include the following:

- **On-Demand Instances** – AWS offers a pay-as-you-go approach for over 70 cloud services, enabling game developers to deploy both quickly and cheaply as their game gains users. Like the utilities that provide power or water, you pay only what you consume, and once you stop using them, there are no additional costs.
- **Reserved Instances** – Some AWS services like Amazon EC2 allow you to enter into a 1 or 3 year agreement in order to gain additional savings on the on-demand cost of these services. With Amazon EC2 in particular, you can choose to pay either no upfront cost for an exchange in reduced hourly cost, or pay all upfront for additional savings over the year (no hourly costs).
- **Spot Instances** – Amazon EC2 Spot Instances enable you to bid on spare Amazon EC2 capacity as a method of significantly reducing your computing spend. Spot Instances are great for applications that are tolerant to workload interruptions; some use cases include batch processing and analytics pipelines that aren't critical to your primary game functioning.

- **Savings Plans** – Savings Plans is a flexible pricing model that provides savings of up to 72% on your AWS compute usage. This pricing model offers lower prices than On-Demand, in exchange for a commitment to use a specific amount of computer power for a one or three year period.
- **Serverless model** – Some other services like AWS Lambda are more granular in their approach to pricing. Instead of being pay-by-the-hour, they are billed in either very small units of time like milliseconds, or by request count instead of time. This allows you to truly pay for only what you use, instead of leaving a service up, but idle and accruing costs.

Conclusion and next steps

We've covered a lot of ground in this book. Let's revisit the major takeaways and some simple steps you can take to begin your game's journey on AWS:

- Start simple, with two EC2 instances behind an Elastic Load Balancing load balancer. Choose either Amazon RDS or Amazon DynamoDB as your database. Consider using AWS Elastic Beanstalk to manage this backend stack.
- Store binary content such as game data, assets, and patches on Amazon S3. Using Amazon S3 offloads network-intensive downloads from your game servers. Consider CloudFront if you're distributing these assets globally.
- Always deploy your EC2 instances and databases to multiple Availability Zones for best availability. This is as easy as splitting your instances across two Availability Zones to begin with.
- Add caching via ElastiCache as your server load grows. Create at least one ElastiCache node in each Availability Zone where you have application servers.
- As the load grows, offload time-intensive operations to background tasks by using Amazon SQS or another queue such as RabbitMQ. This enables your EC2 app instances and database to handle a higher number of concurrent players.
- If database performance becomes an issue, add read replicas to spread the read/write load out. Evaluate whether a NoSQL store such as DynamoDB or Redis could be added to handle certain database tasks.
- At extreme loads, advanced strategies such as event-driven servers or sharded databases may be necessary. However, wait to implement these until necessary, since they add complexity to development, deployment, and debugging.

Finally, remember that Amazon Web Services has a team of business and technical people dedicated to supporting our gaming customers. To contact us, fill out the form at the AWS Game Tech website.

Contributors

Contributors to this document include:

- Greg McConnel, Sr. Manager, AWS Security and Identity Compliance
- Keith Lafaso, Sr. Technical Account Manager, AWS Enterprise Support
- Chris Blackwell, Sr. Software Development Engineer, AWS Marketing

Further reading

For additional information, see:

- AWS Game Tech website
- AWS Marketplace
- AWS Support
- AWS Architecture Center
- AWS Whitepapers & Guides
- AWS Documentation

Blog Posts and Articles

- Best Practices in Evaluating Elastic Load Balancing
- Top 10 Performance Tuning Tips for Amazon Athena
- Best Practices for Amazon EMR
- Fitting the Pattern: Serverless Custom Matchmaking with Amazon GameLift
- Performance at Scale with Amazon ElastiCache
- MongoDB on AWS: Guidelines and Best Practices

Document revisions

Date	Description
December 2019	First publication
March 11, 2021	Reviewed for technical accuracy