全面剖析 AWS 上的可扩展游戏模式

2017 年 9 月

通告

本文档所提供的信息仅供参考，且仅代表截至本文件发布之日时 AWS 的当前产品与实践情况，若
变更恕不另行通知。客户有责任利用自身信息独立评估本文档中的内容以及任何对 AWS 产品或服
的使用方式，任何“原文”内容不作为任何形式的担保、声明、合同承诺、条件或者来自 AWS 及
附属公司或供应商的授权保证。AWS 面向客户所履行之责任或者保障遵循 AWS 协议内容，本文
与此类责任或保障无关，亦不影响 AWS 与客户之间签订的任何协议内容。

目录

概述

本份白皮书旨在对 AWS 各相关服务作出剖析，探讨这些服务如何帮助游戏行业之内的架构师、开
者、运营人员（IT 与 DevOps）以及技术领导者面向常规游戏工作负载——包括内容更新、分析
游戏服务器以及其它数字化服务——构建起可扩展、可靠且具备成本效益的云解决方案。

尽管本份白皮书主要面向尚未接触过 AWS 的新客户，但经验老道的 AWS 用户同样能够从中发现
合自身的有价值内容。

内容简介

无论您是一位刚刚投身相关行业的移动开发人员，抑或效力于 3A 级游戏工作室，大家都有必要充了解在目前的游戏行业之内打造出成功游戏作品时所面临的实际挑战。除了游戏本身必须拥有突亮点之外，用户亦期待您为其提供广泛的在线功能——包括好友列表、排行榜、每周挑战、各类人游戏模式以及内容持续发布等等。为了成功推出游戏产品，我们首先需要在应用商店中赢得正的评论与评分结果——正如一部新近上映的电影，其后续命运往往取决于首周末票房水平与宣传度一样。为了实现上述功能，您需要一套服务器后台。该服务器后台可以作为用于支持多人对战游戏服务器，亦可充当在线聊天、玩家匹配等游戏服务功能的平台服务器。这些服务器后台必须能力在必要时实现横向扩展，例如游戏因质量出色而一炮而红，并使得用户数量由 100 位快速增至 10 万位的情况。与此同时，这些后台还必须具备成本效益，确保大家不需要为未经充分利用的务器容量而买单。

Amazon Web Services (简称 AWS) 是一套灵活、具备成本效益且易于使用的云平台。通过将您的戏运行在 AWS 之上，大家可以根据需求配合用户趋势对容量规模进行伸缩，而不必被迫猜测服务需求并面临潜在的硬件过度采购或者采购不足等问题。众多社交、移动乃至 3A 级开发商都已经意到 AWS 的突出优势，并成功将其游戏运行在 AWS 云之上。

本份白皮书在内容上分别涵盖现代游戏当中的不同功能，具体包括好友列表、排行榜、游戏服务器消息收发以及用户生成内容等等。大家可以从小处着手，根据需求选用各 AWS 组件与服务。随着的游戏不断发展壮大，您也可以重新回顾这份白皮书以进一步评估其它更适合当前需求的 AWS 功

AWS 拥有一支商务与技术支持团队，其致力于服务我们的游戏行业客户。如果您对于本份白皮书内容或者在 AWS 之上运行游戏抱有任何疑问，请马上与我们直接联系。请点击此处并填写网站中 AWS 游戏开发商表单，这也许会成为我们良好合作的理想开端。[1]

从这里开始

如果您刚刚开始开发自己的游戏作品，那么找到最适合自己的后台服务器开发方案往往颇具挑战
值得庆幸的是，AWS 能够帮助大家快速上手，且无需在确定实际需求之前匆忙作出任何服务器决
在对游戏进行迭代时，大家可以随时间推移逐步添加 AWS 服务。如此一来，您将无需在开发初期
一切作出规划，即可顺利向游戏当中添加新特性或者后台功能。我们鼓励您根据游戏初期的功能
求选择服务项目，而后随游戏发展而引入更多 AWS 服务。在本章节中，我们将介绍部分常见游戏
能，希望帮助大家确定您所需要的基本服务类型。

游戏设计决策

现代社交、移动与 3A 级游戏倾向于采取以下几项共通性服务器架构设计思路：

- **排行榜与排名** – 游戏玩家将获得不断推进的竞争性体验，这一点与经典的街机游戏较为相
但更重要的是，好友间排行榜的加入不再单纯提供全球高分排名。而这要求我们构建更为复杂的
样机制，其必须能够立足多种维度进行排序，同时保持良好的性能表现。

- **免费游玩** – 过去几年以来，游戏领域的最大变化在于面向免费网络游戏模式的快速转型。
这套模式当中，游戏本身可供免费下载与游玩，但提供内购系统通过向玩家销售武器、服装、升
点数以及技能点数等物品获利——此外亦存在通过广告收益获利的方式。这意味着您的游戏后台
要尽可能提升性价比水平，且具备规模扩展能力并可根据实际需求进行规模伸缩。就目前而言，
使是大比例收益由首发销售贡献的 3A 级游戏，也开始越来越多地实行付费内容更新与游戏内购机

- **分析** – 要最大程度实现长期营收目标，我们需要在游戏内建立收集机制并分析与游戏体验
式、购买偏好等核心要素相关的大量指标。为了确保新的游戏特性始终有助于吸引玩家在其中投
时间与金钱，成功的游戏内购元素无疑至关重要。

- **内容更新** – 要实现最理想的玩家挽留率，游戏必须通过持续发布周期放出更多新物品、关卡
挑战以及成就。游戏产品的这种持续开发趋势使其更像是一种需要在发布之后不断变化的服务项
这些功能需要配合新数据与游戏资产通过频繁更新来实现。而通过使用内容交付网络（简称 CD
进行游戏内容分发，您将能够显著节约成本并提升玩家的下载速度。
- **异步游戏体验** – 尽管大型游戏往往包含实时在线多人游戏模式，但目前各类游戏作品如今
开始越来越多地意识到异步功能的重要性——特别是在保持玩家参与度方面。举例来说，异步游
体验包括用户与好友间的分数、解锁数量、徽章收集或者其它类似成就的收集竞争。这意味着即
无法持续保持在线，或者需要使用 3G 或者 4G 等速度较慢的移动网络进行游戏，玩家们也仍然能
获得连续不断的在线式游戏体验。
- **推送通知** – 让用户时常回到游戏中看看的常用方法之一，是由游戏向用户的移动设备定期
布针对性推送消息。举例来说，用户可能会得到通知称其某位好友已经超过了由其保持的最高分
者开发商发布了可供游玩的新挑战及关卡。如此一来，即使玩家当前并未进行游戏，也很可能被
引并重新回归开发者为其设计的核心游戏体验。
- **不可预知的客户** – 现代游戏运行在各类设备平台之上，包括移动设备、游戏主机、PC 以及
览器等等。用户能够以漫游方式通过 Wi-Fi 在自己的移动设备上同游戏主机玩家对战，且双方皆
获得一致的游戏体验。为了达成这一目标，我们必须尽可能使用无状态协议（例如 HTTP）与异
调用等手段。

以上提到的每一项游戏功能都会对您的服务器特性与技术产生影响。举例来说，如果您希望设计
套简单的前十名排行榜，则可以直接将其存储在单一 MySQL 或者 Amazon Aurora 数据库表当
然而如果您需要打造一套包含多种序列维度的复杂排行榜，则可能必须利用 Redis 或者 Amazon
DynamoDB 等 NoSQL 技术方案（我们将在后文中详加讨论）。

游戏客户端考量

尽管本份白皮书将主要着眼于您在 AWS 进行部署时的实际架构，但您的游戏客户端也会在另一方
对游戏本体的可扩展性造成重大影响。另外，游戏客户端也在一定程度上决定着游戏后台的运行
本——这是因为来自客户端的频繁网络请求会占用更多带宽以及服务器资源。以下为需要关注的
项重要指导原则：

- 所有网络调用都应当以异步及非阻塞方式进行。这意味着当某一网络请求启动时，游戏客
端仍在继续运行而无需等待来自服务器的响应。当服务器作出响应后，客户端会触发相应事件—
此事件由客户端代码中的某种回调机制负责处理。在 iOS 系统上，AFNetworking 为比较流行的
案之一。[2] 浏览器游戏则应使用 jQuery.ajax()[3] 或者其它类似方案进行调用，而 C++客户端可以考
libcurl[4]、std::async[5] 或者其它类似的库选项。

- 利用 JSON 进行数据传输。其结构紧凑、具备跨平台能力、解析速度极快、拥有大量支持
选项且包含有数据类型信息。如果大家拥有规模可观的载荷，则可直接利用 gzip 对其进行压缩，
为目前大多数 Web 服务器与移动客户端都原生支持 gzip。另外，不要浪费时间进行过度优化——
何不超过数百 KB 大小的载荷都应归于可接受范畴。当然，我们也发现一部分开发者会在用例当
选择 Apache Avro 或者 MessagePack，他们更喜爱这些格式以及可供选择的相关库。备注：多人
戏数据包属于例外，其算是典型的 UDP，并非本章节打算讨论的内容。

- 使用 HTTP/1.1 配合 Keepalives，并复用各请求之间的 HTTP 连接。这能够在确保网络请
得到执行的同时，有效削减游戏运行开销。每当您需要开启新的 HTTP 套接，其会要求进行一次
方 TCP 握手，并由此将延迟水平增加约 50 毫秒。另外，重复开启与关闭 TCP 连接会在您服务器
TIME_WAIT 状态中累积大量套接，进而消耗掉极具价值的服务器资源。

- 始终将来自客户端的重要数据经由 SSL POST 至服务器。其中具体包括登录、状态、保存
据、解锁以及购买等。同样的要求亦适用于 GET、PUT 以及 DELETE 等请求，这是因为现代计
机能够高效处理 SSL，意味着相关开销极低。AWS 允许大家使用我们的弹性负载均衡器（Elastic Lo
Balancer）处理此类 SSL 工作负载，从而帮助您的服务器回避相关处理负担。[6]

- 永远不要在客户端设备上存储 AWS 访问密钥或者其它令牌等关键性安全数据，亦不要在
中存储您的游戏数据或者用户数据。访问密钥 ID 与秘密访问密钥允许这些密钥的拥有者直接经
AWS 命令行界面（简称 AWS CLI）、Tools for Windows PowerShell、AWS SDK 或者直接 HTTP
用（利用 API 调用独立 AWS 服务）面向 AWS 执行编程式调用。如果用户对其设备进行了 root
者越狱，则您有可能面临着服务器代码、用户数据甚至是您自己的 AWS 计费帐户遭到外泄的风
在 PC 游戏方面，您的密钥可能会在游戏客户端运行时存在于内存当中，并在游戏客户端关闭时
其清除。作为开发者，大家必须假设您存储在游戏客户端当中的数据已经遭到窥探。如果您希望
己的游戏客户端直接访问 AWS 服务，请考虑使用 Amazon Cognito Federated Identities，其允许
的应用获取临时性、受限权限凭证。[7]

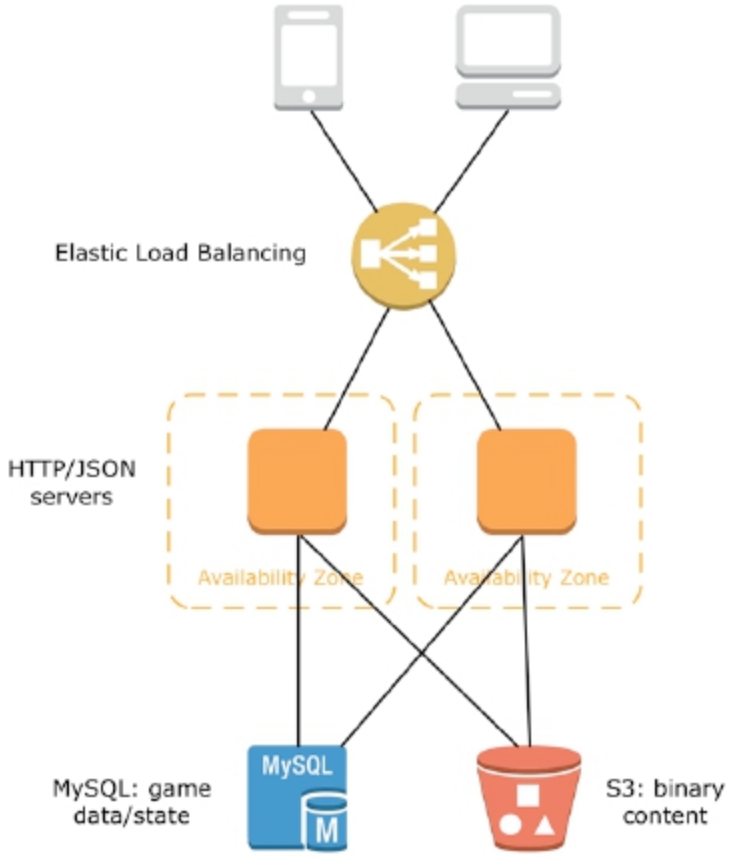
- 作为另一项预防措施，请永远不要信任游戏客户端发送给您的数据。其属于不受信来源，
此大家应当始终对收到的数据进行验证。这类数据有时属于恶意流量（包括 SQL 注入与 XSS 等等
但有时候只是用户对设备时间进行调整后生成的记录。
这里要再次强调，虽然以上提到的部分问题并非 AWS 所特有，但仍然建议大家在进行游戏设计时
虑到这些因素——这将帮助您确保自己的游戏作品运行良好且安全可靠。

启动初始游戏后台

在考虑到以上游戏功能与客户端注意事项之后，下面我们将探讨如何启动初始游戏后台并将其尽
运行在 AWS 之上。在这一过程当中，我们会用到一些关键性 AWS 服务，并充分发挥其随游戏激
而添加更多功能与服务的能力。

为了确保能够随着游戏人气的增长对其进行向外扩展，我们还将尽可能使用无状态协议。通过为
类游戏功能创建对应的 HTTP/JSON API，我们将能够以动态方式添加实例并轻松从网络传输问
当中恢复正常。我们的游戏后台包含一台与 HTTP/JSON 对话的服务器、用于存储数据的 MySQ
以及用于存放二进制内容的 Amazon 简单存储服务（Amazon Simple Storage Service，简称 Amaz
S3）。这类后台易于开发且能够高效完成规模伸缩。

对于游戏开发者而言，最常见的开发模式是利用笔记本电脑或者台式机运行一套本地 Web 服务而后将服务器代码推送至云端以进行实际部署。如果大家采取这样的模式，那么 AWS Elastic Beanstalk 将能够极大简化您将代码部署至 AWS 的流程。[8]



图一：高层概述——在 AWS 上运行您的第一套游戏后台

Elastic Beanstalk 是一项部署管理服务，其立足于 Amazon 弹性计算云（Amazon Elastic Compu Cloud，简称 Amazon EC2）、弹性负载均衡（简称 ELB）以及 Amazon 关系数据库服务（Amaz Relational Database Services（简称 Amazon RDS）等其它 AWS 服务之上。

Amazon EC2 是一项负责在云端提供安全且可伸缩的计算容量的 Web 服务。其主要作用在于帮助 发者更轻松地实现云计算规模化使用。Amazon EC2 简单 Web 服务界面允许您以最轻松易行的方 实现容量获取与配置。其能够将新服务器实例的获取与启动时长缩短至数分钟以内，帮助大家快 根据计算需求变化进行容量规模伸缩。

ELB 能够跨越多个 Amazon EC2 实例自动分发输入应用流量。其将帮助用户在您的应用程序之内现容错能力。ELB 提供两种负载均衡器类型，可保证高可用性、自动规模伸缩与强安全性要求。中的经典负载均衡器基于应用程序或者网络层级信息进行流量路由，而应用负载均衡器则基于其高级应用层级信息进行流量路由——具体包括请求内容。经典负载均衡器较适用于简单的跨多 E实例流量负载均衡，而应用负载均衡器则更适用于需要高级路由能力、微服务以及基于容器型架的应用程序。

Amazon RDS 能够在云端轻松实现关系数据库的设置、运营与规模调整。其在提供极具性价比与伸缩容量的同时，亦能够自动完成硬件配置、数据库设置、补丁安装以及备份等相当耗时的管理务。Amazon RDS 支持六种广受欢迎的数据库引擎，具体包括 Amazon Aurora、PostgreSQL、MySQ MariaDB、Oracle 以及微软 SQL Server。

大家可以面向 Elastic Beanstalk 推送 zip、war 或者 git 库格式的服务器代码。Elastic Beanstalk 责启动各相关 EC2 实例，添加负载均衡器，创建 RDS MySQL 数据库实例，设置 Amazon CloudWa监控警报，并将您的应用程序部署至云环境当中。简而言之，Elastic Beanstalk 能够自动设置如一所示之架构，感兴趣的朋友可以参阅 Elastic Beanstalk 架构以了解更多细节信息。[9]

为了了解具体操作方法，大家需要登录至 AWS 管理控制台 [10] 并遵循 Elastic Beanstalk 上手指南教程的说明利用您所选定的编程语言创建一套新环境。其中具体包括启动示例应用程序并引导一默认配置。您可以利用这套环境体验 Elastic Beanstalk 控制面板、如何更新代码以及如何修改环设置等等。如果大家刚刚接触 AWS，则可使用 AWS 免费层（AWS Free Tier）设置这些示例环[12]

备注: 本份白皮书当中描述的示例生产环境会带来实际使用成本，因为其中囊括部分超出免费层范围
收费 AWS 资源。

在示例应用程序当中，我们将为自己的游戏创建一款新的 Elastic Beanstalk 应用，其中囊括两套新
境——其一用于开发，其二则用于生产。我们将对两套环境作出些许定制化调整，以确保其符合我
游戏的实际需求。利用以下表格，我们可以检查需要根据环境类型加以变更的设置。欲了解更多细
说明，请参阅 AWS Elastic Beanstalk 应用程序管理与配置 [13]，同时查看 AWS Elastic Beanstalk 开
者指南 [14] 当中的创建一套 AWS Elastic Beanstalk 环境。

备注: 请注意将我的游戏与 mygame 值替换为您游戏的实际名称。

配置设定	开发值	生产值
应用程序名称	我的游戏	我的游戏
环境名称	mygame-dev	mygame-prod
环境 URL	http://mygame-dev.elasticbeanstalk.com	http://mygame-prod.elasticbeanstalk.com
容器类型	64 位	64 位
实例类型	t2 套微型或小型实例	m4.medium 或 c4.large
运行状态检查 URL		/healthchk (或其它类似方法)
是否创建 RDS DB 实例?	是	是
数据库引擎	mysql	**不推荐
实例类	db.t2.small 或 db.t2.medium	N/A
存储资源分配	5 GB	N/A
删除策略	删除	创建快照 t
是否使用多可用区?	否	是

通过使用这两套环境，大家可以启动一套简单而高效的工作流程。随着您不断整合新的游戏后台
能，您将可把经过更新的代码推送至开发环境当中。这类操作将触发 Elastic Beanstalk 重启整体
境并创建新版本。在您的游戏客户端代码当中，请创建两套配置方案——其一指向开发环境，其
指向生产环境。利用开发配置测试您的游戏，而后利用生产配置将创建完成的新游戏版本发布至
想的应用商店当中。

当您的新游戏客户端作好发布准备之后，请从开发环境中选择正确的服务器代码版本，并将其部
至生产环境当中。在默认情况下，部署工作会带来短暂的停机时间——这是因为您的应用程序正
更新并重新启动。如何希望避免部署操作带来的停机问题，大家可以采用交换 URL 或者蓝/绿部
机制。在这种模式下，大家可以将代码部署至备用生产环境处，而后更新 DNS 以指向这套新环
与此相关的更多细节信息，请参阅 AWS Elastic Beanstalk 开发者指南当中的利用 AWS Elastic Beanstalk 实现蓝/绿部署相关内容。[15]

重要说明： 我们不建议大家利用 Elastic Beanstalk 管理生产环境下的数据库，这是因为此类作法会将该数
库实例（DB 实例）的生命周期同您应用程序环境的生命周期联系起来。

相反，我们建议您在 Amazon RDS 当中运行一个数据库实例，并通过配置确保您的应用程序在启动时与之对
大家也可以在 Amazon S3 当中存储连接信息，并配置 Elastic Beanstalk 以在利用. ebextensions 进行部署时检查这
分信息。大家可以将 AWS Elastic Beanstalk 配置文件（. ebextensions）添加至您的 Web 应用程序源代码当中，
配置您的环境并定制其中所包含的各项 AWS 资源。这些配置文件为 YAML 格式文档且扩展名为.config，大家可以
放置在名为. Ebextensions 的文件夹当中并部署于您的应用程序源代码包内。

欲了解更多细节信息，请参阅 AWS Elastic Beanstalk 开发者指南当中的利用配置文件（. ebextensions）
高级环境定制化相关内容。

可用性、可扩展性与安全性

对于生产环境，大家需要确保您的游戏后台以容错方式进行部署。Amazon EC2 托管于全球范围
多个 AWS 服务区之内。您应尽可能选择与大多数游戏客户所在位置邻近的服务区，这将确保您的
户在进行游玩时享受到较低延迟水平。欲了解更多与 AWS 服务区相关信息以及最新选项清单相关
内容，请参阅 AWS 全球基础设施页面。[16]

各个服务区之内都存在着多个位置彼此隔离的可用区，大家可以将其视为逻辑数据中心。每个可
区立足特定服务区当中，且彼此保证物理隔离，但通过高速网络进行相互对接——如此一来，客
即可根据需求同时使用多个可用区。通过在同一服务区内的多个可用区间进行服务器负载均衡，
将能够轻松有效地提升游戏的可用性水平。另外，我们建议您使用 2 个可用区进行可靠性与游戏
本均衡，这是因为大家可以借此实现服务器实例、数据库实例以及缓存实例间的配对。

Elastic Beanstalk 能够自动跨越多个可用区实现部署。要配合 Elastic Beanstalk 使用多可用区，
家可以点击此处参阅 AWS Elastic Beanstalk 开发者指南当中的利用 Elastic Beanstalk 配置 Auto
Scaling 一文。[17] 如果要进一步实现可扩展性，您也可以利用 Auto Scaling 随时从这些可用区内添
及移除实例。要获得最佳效果，请考虑修改 Auto Scaling 触发条件以指定一项指标（例如 CPU 使
率）并根据应用程序性能要求设置阈值。如果您所指定的阈值被触及，则 Elastic Beanstalk 会自
启动更多实例。我们将在后文当中的 Auto Scaling 章节当中对此进行深入探讨。

对于开发及测试环境而言，单一可用区通常已经足够，这意味着您可以在能接受的停机故障机率
下获得低廉的资源使用成本。然而，如果您的开发环境需要供 QA 测试人员在深夜使用，那么大
可能需要将其设置得同生产环境更为相似。在这种情况下，使用多可用区将切实满足您的需求。

最后，设置负载均衡器以处理 SSL 终端，SSL 加密与解密负载将不再需要由您的游戏后台服务器
担。相关细节信息请参阅 AWS Elastic Beanstalk 开发者指南当中的为您的 Elastic Beanstalk 环境
置 HTTPS 部分。[19] 我们强烈建议您将 SSL 添加至游戏后台。欲了解更多 ELB 相关技巧，请参阅
份白皮书后文中的 HTTP 负载均衡章节。

利用 Amazon S3 存储二进制游戏数据

最后，大家还需要为每台 Elastic Beanstalk 服务器环境创建一个 S3 存储桶。此 S3 存储桶负责存
您的二进制游戏内容，包括补丁、关卡与资产等等。Amazon S3 利用一个基于 HTTP 的 API 进行
据上传与下载，这意味着您的游戏客户端能够使用同一套 HTTP 库与游戏服务器进行通信，从而
现游戏资产下载。利用 Amazon S3，您只需要为实际使用的数据存储容量以及客户端下载时占用
传输带宽付费。欲了解更多细节信息，请参阅 Amazon S3 计费页面。[19]

首先，我们需要在服务器所在的服务区内创建一个 S3 存储桶。举例来说，如果大家将 Elastic Beanstalk 部署在美国西部 2（俄勒冈州）服务区内，则请选择同一服务区进行 Amazon 部署。为了简单起见，我们在这里为存储桶选择与 Elastic Beanstalk 环境类似的命名方式（例如 mygame-dev 或者 mygame-prod）。欲获取分步指导，请参阅 Amazon S3 开发者指南当中的创建储桶部分。[20] 需要注意的是，请务必为您的每一套 Elastic Beanstalk 环境创建独立的 S3 存储桶（如开发环境与生产环境等）。

在默认情况下，S3 存储桶以私有方式存在，要求用户进行身份验证方可下载其中的内容。对于游内容，大家拥有两种处理选项。第一，将该存储桶彻底公开，意味着任何了解存储桶名称的人士能够借此下载游戏内容。第二种更好的办法，是利用签名 URL 进行验证管理——利用此项功能，将能够把 Amaozn S3 凭证作为访问 URL 中的组成部分。如此一来，您的游戏服务器代码会将用重新定向至某一由 Amazon S3 签名的 URL，而您可以设置特定时间段对相关凭证进行过期处理关于如何创建签名 URL，请参阅 Amazon S3 API 参考资料中的验证请求（AWS 签名版本 4）。如是配合自选语言的官方 AWS SDK，也可以使用 SDK 所内置的预签名 URL 生成功能。预签名 U允许大家访问该 URL 所标识的对象，但前提是该预签名 URL 的创建者有权访问该对象。由于预名 URL 的生成操作完全以离线方式进行（不需要进行任何 API 调用 ），因此执行速度极快。

最后，随着游戏规模的不断增长，大家可以利用 Amazon CloudFront 提供更理想的性能表现，并此帮助自身节约数据传输成本。欲了解更多相关信息，请参阅 Amazon CloudFront 开发者指南中 Amazon CloudFront 是什么部分。[21]

不止于 AWS Elastic Beanstalk

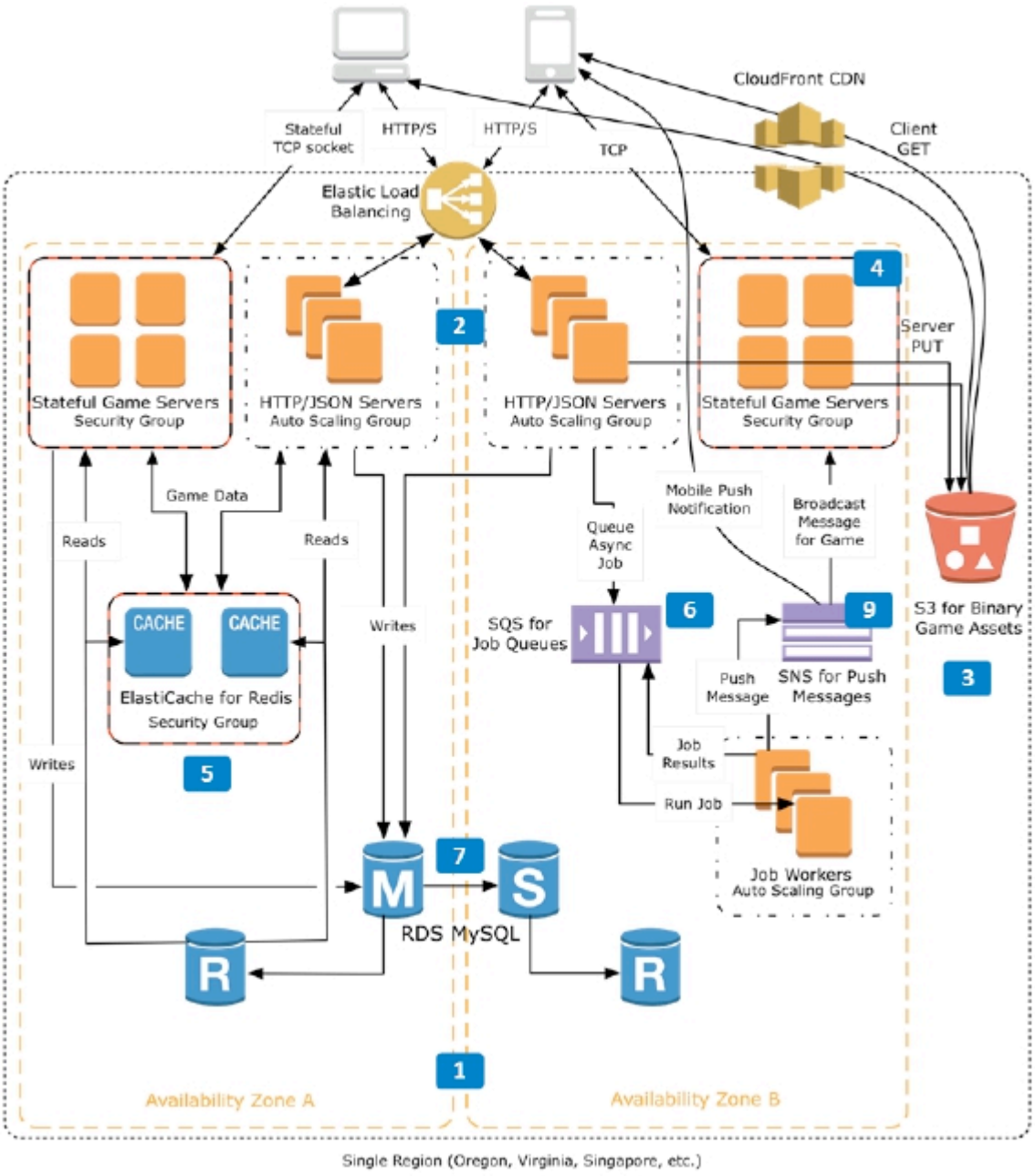
核心游戏后台系统会随着游戏玩家人数的增长而快速扩展，这当然需要我们作出响应以确保用户有一致的性能体验。通过利用 HTTP 处理批量调用，您将能够轻松根据使用模式的变化对响应容进行规模伸缩。而相较于直接通过 Amazon Ec2 进行文件交付，将二进制数据存储在 Amazon S3 能够带来理想的成本优势; 另外，Amazon S3 还能够帮助您打理数据可用性与持久性等问题。Amazon RDS 则提供一套受控 MySQL 数据库，您可以随时间推移不断添加 Amazon RDS 功能，如读取副本等等。

如果您的游戏需要更多其它功能，则可通过扩展引入 Elastic Beanstalk 之外的其它 AWS 服务，且本架构仍能够继续生效。Elastic Beanstalk 支持通过 Elastic Beanstalk 环境资源对其它 AWS 服务行配置。[22] 举例来说，大家可以利用 Amazon ElastiCache 添加一个缓存层，这项托管缓存服务同支持 Memcached 与 Redis。欲了解更多与添加 LeastiCache 集群相关的细节信息，请参阅 Amazon ElastiCache 用户手册当中的示例片段：ElastiCache 部分。[23]

当然，大家也可以单独使用其它 AWS 服务，并配置应用程序以运用相关资源。举例来说，您可以择添加或者替换自己的 RDS MySQL 数据库实例或者 Amazon Aurora 数据库，替换选项包括 DynamoDB、AWS 托管 NoSQL 方案或者 MongoDB 等其它 NoSQL 数据库。即使您最初是以 Elastic Beanstalk 作为起点，也仍然可以随时间推移与规模增长而不断引入其它 AWS 服务。

参考架构

在我们的核心游戏后台上线并开始运行之后，接下来就是检查其它可能对游戏表现有所帮助的 AWS 服务。在持续这一话题之前，让我们首先看看以下参考架构——其主要面向一套横向可扩展游戏后台。示意图中描绘了相当广泛的游戏后台功能，具体包括登录、排行榜、挑战、聊天、游戏二进制数据、用户生成内容、分析以及在线多人模式等等。诚然，并非所有游戏都拥有下列全部功能，但这份图表能够帮助我们清晰理解如何将这些组成部分结合起来。在本份白皮书的剩余内容中，我们将对其中各组成部分展开深入探讨。



图二：运行在 AWS 之上的一套生产就绪型游戏后台

图二初看起来似乎有些复杂，但其实际上就是我们最初利用 Elastic Beanstalk 启动的游戏后台的变产物。

编号	说明
-	本示意图中显示 2 个可用区，设置有相同功能以实现冗余。尽管空间所限，我们没有将所有组成部分皆纳入图中，但可肯定的是 2 个可用区确实能够等效运作。这些可用区可以与您最初利用 Elastic Beanstalk 时选定的可用区完全一致。
-	HTTP/JSON 服务器与主/从数据库可与您此前利用 Elastic Beanstalk 启动的服务器与数据库完全相同。您可以尽可能独立足 HTTP/JSON 层构建更多游戏功能。您可以利用 HTTP Auto Scaling 根据用户需求自动添加及移除 EC2 HTTP 实欲了解更多相关信息，请参阅本份白皮书中的 HTTP Auto Scaling 章节。
-	您可以使用初始为二进制数据所创建的同一 S3 存储桶。Amazon S3 具备出色的可扩展性，且随时间推移几乎不需要调优。随着您游戏资产与用户流量的持续增加，大家可以在 Amazon S3 之前添加 Amazon CloudFront 以强化下载性节约运营成本。
-	如果您的游戏需要使用有状态套接，例如借此实现聊天或者多层游戏体验，那么这些功能通常由负责运行代码的游戏服处理。这些服务器运行在多个 EC2 实例之上，且与您的 HTTP 实例区分开来。欲了解更多细节信息，请参阅本份白皮文中的有状态游戏服务器章节。
-	随着游戏规模的增长与数据库负载的提升，接下来大家需要添加缓存机制——通常利用 Amazon ElastiCache 实现，这项 AWS 托管缓存服务。ElastiCache 缓存会接过原本指向数据库的读取查询重担。本份白皮书将在之后的部分中对缓制作出详尽说明。
-	下一步在于将服务器上的部分任务转化为异步作业，这方面工作可以利用 Amazon 简单队列服务（Amazon Simple Q Service，简称 SQS）实现 [24]。如此一来，您将能够拥有一套松散耦合架构，存在多种组件，各组件几乎完全意识不到组件的存在，但却能够互操作以实现特定目的。Amazon SQS 确保松散耦合系统中的各组件不再相互依赖。举例来说果您的游戏允许用户上传并共享照片或者自定义头像等资产，则应利用一项后台作业执行这类相当耗时的图像大小调整这不仅能够让您的游戏拥有更快的响应速度，同时也能够降低您 HTTP 服务器实例所面对的负载强度。我们将在本份书后文的异步作业章节中对此作出深入探讨。
-	随着您数据库负载的持续增长，大家可以添加 Amazon RDS 读取副本以帮助进一步扩展数据库读取规模。这种作法亦于降低主数据库的负载强度，因为我们可以借此将部分读取操作分流至副本处，而仅针对主数据库进行写入操作。我们本份白皮书的 MySQL 章节部分对此作出深入探讨。
-	与此同时，您还应决定是否引入 Redis 或者 DynamoDB 等 NoSQL 服务，借以补充主数据库的排行榜等具体功能; 或者用原子计数器等 NoSQL 功能。我们将在本份白皮书中的 NoSQL 数据库章节内对这些选项展开进一步探讨。（备注：现在图二当中。）
-	如果您的游戏当中包含通知推送机制，则可使用 Amazon 简单通知服务（Amazon Simple Notification Service，简 Amazon SNS）[25] 及其 Mobile Push 支持 [26] 以简化跨移动平台发送推送消息的流程。您的 EC2 实例亦可接收 Amazon SNS 消息，帮助大家实现诸如面向所有正在接入游戏服务器的玩家发布广播信息的能力。

如果着眼于图二中的单一可用区，并将其与我们利用 Elastic Beanstalk 启动的核心游戏后台进行
较，大家就会意识到该如何通过添加缓存、数据库副本以及后台作业等方式立足初始后台组件进
游戏规模扩展。考虑到这一点，下面我们将对其中各组件进行深入探讨。

游戏即 REST APIs

如之前所提到，要实现横向扩展，大家应利用 HTTP/JSON API 实现大部分游戏功能——而 HT
JSON API 通常遵循 REST 架构模式 [27]。无论运行在移动设备、平板电脑、PC 抑或是游戏主机之上
游戏客户端都会面向服务器发出与数据相关的 HTTP 请求，具体包括登录、会话、好友、排行榜
至成就奖杯等等。客户端不会长期与服务器端保持连接，因此我们可以通过添加 HTTP 服务器实
的方式轻松完成横向扩展。各客户端亦可通过简单重试 HTTP 请求从网络问题当中恢复过来。

在设计合理的情况下，REST API 应该能够扩展并支持数十万名玩家同时在线。我们在此前讨论
Elastic Beanstalk 示例时也一直以此为前提。RESTful 服务器能够轻松部署在 AWS 之上，且允许
家直接利用 AWS 上所提供的各类 HTTP 开发、调试以及分析工具。

但也有一部分游戏模式更适合采用有状态双工套接机制，从而接收由服务器发起的消息发送。相
实例包括实时在线多人游戏、聊天或者游戏邀请等等。如果您的游戏不具备这些功能，则可放心
用 REST API。我们将在本份白皮书的后文部分讨论有状态服务器，但这里我们姑且专注于 RES
层面。

要面向 Amazon EC2 部署一个 REST 层，通常需要一套 Nginx 或者 Apache HTTP 服务器，外加
套特定语言应用服务器。以下表格列出的正是目前游戏开发者们经常用于构建 REST API 的流行
件包选项。

语言	软件包
Ruby	Rails,[28] Sinatra,[29] Grape[30]
Python	Flask,[31] Bottle[32]

语言	软件包
Node.js	Express,[33] Restify[34]
PHP	Slim,[35] Silex [36]
Java	Spring,[37] Jersey[38]
Go	Gin[39]

以上只是示例内容，您可以利用任何适合 Web 开发的编程语言构建 REST API。由于 Amazon E 允许大家以完整 root 权限访问实例，因此您可以部署以上任意软件包。对于 Elastic Beanstalk，支持软件包数量则较为有限。欲了解更多细节信息，请参阅 Elastic Beanstalk 常见问题解答。[40]

RESTful 服务器适合选用中等大小的实例，这是因为客户能够借此以同样的价格部署更多横向扩 示例。中等大小实例在通用型实例家族（例如 M4）或者计算优化型实例家族（例如 C4）中都相 常见，且能够良好满足 REST 服务器的实际需要。

HTTP 负载均衡

对 RESTful 服务器进行负载均衡非常简单，这是因为 HTTP 连接以有状态形式存在。AWS 提供 性负载均衡（简称 ELB）服务，其也正是在 Amazon EC2 上实现游戏 HTTP 负载均衡的最简单方案[41] 大家可能还记得，我们在之前的游戏后台示例当中曾利用 Elastic Beanstalk 自动部署 ELB 负载 衡器，并由其负责对各 EC2 实例进行负载均衡。如果大家从起步阶段即使用 Elastic Beanstalk，现在应该已经运行有 ELB 负载均衡器了。

要发挥 ELB 的全部潜能，请尽量遵循以下指导原则：

- 始终立足至少两个可用区配置 ELB，借以实现冗余与容错能力。ELB 会在您所指定的各可 区与具体 EC2 实例之间处理流量均衡工作。如果大家希望将流量平均分配在各服务器上，则应启 跨区负载均衡机制——即使各可用区上的服务器数量有所区别。通过这种方式，我们将能够确保 整套体系内的各服务器加以充分利用。

- 配置 ELB 以处理 SSL 加密与解密。这将帮助 HTTP 服务器摆脱 SSL 相关处理负担，意味能够为您的应用程序代码腾出更多 CPU 资源。欲了解更多信息，请参阅经典负载均衡器指南当中创建一套 HTTPS 负载均衡器部分。[42] 欲在开发流程当中对 SSL 进行测试，则请参阅 AWS 证书管器用户指南中的如何创建一份自签名 SSL 证书内容。[43]

- ELB 会自动从其负载均衡池当中移除一切发生故障的 EC2 实例。为了确保您的 HTTP EC实例拥有良好的运行状态，您需要对其进行精确监控——即利用一条自定义运行状态检查 URL 进负载均衡器配置。在此之后，开发者可编写服务器代码以响应该 URL 并对应用程序的运行状态中检查。举例来说，大家可以设置一项简单的运行状态检查以验证是否具备数据库连接能力。如果行状态检查通过，则其返回 200 Ok; 若实例运行状态不佳，则返回 500 服务器错误。

- 您所部署的每一套 ELB 负载均衡器都拥有一个惟一的 DNS 名称。为了为您的游戏设置一定制化 DNS 名称，大家需要使用一条 DNS 昵称（CNAME）将您游戏的域名指向该负载均衡器欲了解更多细节信息，请参阅经典负载均衡器指南当中的为您的经典负载均衡器配置定制化域名分。[44] 需要注意的是，当您的负载均衡器进行规模伸缩时，该负载均衡器所使用的 IP 地址也会发变化，因此我们必须确保始终使用指向 该负载均衡器的 DNS CNAME 昵称——即避免直接引用 D域名内的负载均衡器当前 IP 地址。

- ELB 在设计当中每 5 分钟进行一次因子约为 50%的等比向上扩展。对于大多数游戏，这一计能够带来理想效果，即足以承载突如其来的人气增长。然而如果大家的游戏因为发布新的下载容或者营销活动而可能面临更为迅猛的流量提升，那么也可对 ELB 进行预热以提前为此作出准要对 ELB 进行预热，我们首先需要利用预期负载提交一份 AWS 支持请求（要求客户至少拥享有业级支持服务[45]）。欲了解更多与 ELB 预热以及针对 ELB 运行负载测试的最佳实践信息，请参阅 AELB 评估最佳实践文章。[46]

欲了解更多与 ELB 相关的细节信息，请参阅弹性负载均衡是什么？相关内容。[47]

应用负载均衡器

应用负载均衡器属于第二代负载均衡器，其能够基于 HTTP/HTTPS 层对流量路由机制进行更为化地控制。除了在之前章节当中提到过的功能之外，应用负载均衡器还能够在游戏类工作负载当带来以下重要助益：

- 明确支持 **Amazon EC2 容器服务(简称 Amazon ECS)** – 通过配置，应用负载均衡器可跨越单一 EC2 之上多个端口实现容器负载均衡。动端口可在 ECS 任务定义当中进行指定，其将在 EC2 实例进行容器调度时为其分配一个非使用端

- **HTTP/2 支持** – 作为旧有 HTTP/1.1 协议的修订版，HTTP/2 与应用负载均衡器能够配合来共同充当二进制协议——而非文本协议——实现网络性能提升。二进制协议本身拥有更高的处效率与更低错误率，因此可以有效提升稳定性水平。另外，HTTP/2 支持复用，这意味着我们能重复使用 TCP 连接以下载来自多个来源的内容，并借此降低网络负担。

- 原生 **IPv6 支持** – 随着 IPv4 地址即将消耗殆尽，许多应用程序供应商开始转换思维，甚至绝不支持 IPv6 的应用程序发布在其平台之上。而应用负载均衡器能够原生支持 IPv6 端点，并实面向 VPC IPv6 地址的路由能力。

- **WebSockets 支持** – 与 HTTP/2 类似，应用负载均衡器同样支持 WebSocket 协议，其允在客户端与服务器之间设置一条长期存在的 TCP 连接。这是一种在效率上远超标准 HTTP 连接的法，通过长时间维持开放心跳的方式节约网络流量。WebSocket 非常适用于交付动态数据，具体括持续更新的排行榜等等，且保证尽可能节约移动设备的网络流量与电量使用水平。ELB 可将监程序由 HTTP 转向 TCP，从而支持 WebScockets。在 TCP 模式下时，ELB 允许建立连接时添加 Upgrade 标头，由 ELB 负载均衡器中止一切闲置时间超过 60 秒的连接（例如在这一时间范围内被发出的数据包）。

这意味着如果 ELB 负载均衡器发送一条升级请求并与其它后台实例建立起 WebSocket 连
则客户端必须重新建立连接，否则 WebSocket 协商将失败。

定制化负载均衡器

或者，您也可以在 Amazon EC2 之上部署自己的负载均衡器，从而满足对 ELB 无法提供的特定功
或者指标的需求。目前较为主流的方案选项包括 HAProxy[48] 以及 F5 公司的 BIG-IP Virtual Editio
——二者皆能够运行在 Amazon EC2 之上。如果您决定使用定制化负载均衡器，请遵循以下建议

- 将负载均衡器软件（例如 HAProxy）部署在两套 EC2 实例之上，每套实例源自不同可用
以实现冗余。

- 为每套实例分配一个弹性 IP 地址。创建一条 DNS 记录，其中容纳有作为您入口点的弹性
地址。如此一来，DNS 将能够在您的两套负载均衡器实例之间轮询。

- 如果您在使用 Amazon Route 53——我们的高可用性、可扩展性云域名系统（简称 DNS）
请配合 Route 53 运行状态检查以监控您的负载均衡器 EC2 实例 [50]。这将确保流量不会被路由至
故障而下线的负载均衡器处。

- 为了利用 HAProxy 处理 SSL 流量，您需要使用最新的 1.5 或者更高开发版本。欲了解更多
息，请参阅 Ilya Grigorik 博客当中的利用 HAProxy 轻松实现 SPDY 与 NPN 谈判一文。[51]

如果您决定部署自己的负载均衡器，请注意另一些同样需要由您自行完成的工作。首先，如果您
负载水平超出了负载均衡器的处理能力，您需要启动更多 EC2 实例并遵循前文提到的步骤将其添
至您的应用程序堆栈当中。另外，新近完成规模自动伸缩的应用程序实例不会被自动注册至您的
载均衡器实例处。大家需要编写一套脚本来更新负载均衡器配置文件并重启各负载均衡器。

如果您打算使用托管服务版本的 HAProxy，则可选择 AWS OpsWorks——其利用 Chef Automa
管理各 EC2 实例，且可将 HAProxy 作为备选方案部署至 ELB 当中。[52]

HTTP Auto Scaling

以动态化方式响应用户模式以实现服务器资源伸缩的能力正是 AWS 云服务的核心优势之一。Au Scaling 允许大家对来自单一或者多个可用区的 EC2 实例进行规模伸缩，从而确保其与 CPU 使用 或者网络吞吐量等系统指标保持一致。欲了解 Auto Scaling 所能提供的完整功能，请参阅 Auto Scaling 是什么？[53] 一文，而后参考 Auto Scaling 上手指南文章。[54]

大家可以利用 Auto Scaling 处理任何 EC2 实例类型，具体包括 HTTP、游戏服务器或者后台工作 序。HTTP 服务器最易于实现规模伸缩，这是因为其处于负载均衡器之后，因此能够由负载均衡 将请求分发至各服务器实例。Auto Scaling 将以动态方式处理来自 ELB 的 HTTP 实例的注册或注 任务，这意味着与之对应的部分流量将很快被指向至新实例处。

要高效使用 Auto Scaling，大家需要选择理想的指标以触发规模扩展与规模收缩操作。为了确定 类指标，请遵循以下指导原则：

- CPU 利用率是一项非常重要的 CloudWatch 指标。Web 服务器往往在 CPU 资源层面有所 制，而内存则在服务器进程的运行过程中相对保持恒定。CPU 利用率的百分比越高，则表明服务 超载的可能性越大。要进一步细化追踪效果，我们还可以将 CPUUtilization 与 NetworkIn 或者 NetworkOut 等指标结合起来。

- 对服务器进行基准测试，用以确定规模伸缩值。对于 HTTP 服务器，大家可以使用 Apach Bench[55] 或者 HTTPerf[56] 等工具衡量您服务器的响应时间。接下来，提升服务器负载并继续监控 C 或者其它指标。需要注意的是，一旦您的服务器响应时间发生降级，马上将其记录下来并尝试把 果与其它系统指标相关联。

- 在对您的 Auto Scaling 组进行配置时，请选择 2 个可用区以及至少 2 台服务器。这将确保 的游戏服务器实例跨越多可用区实现理想的高可用性水平。ELB 将帮助我们打理 2 个可用区之间 负载均衡任务。

欲了解更多与规模伸缩策略相关的配置信息，请参阅 Auto Scaling 用户指南中的[根据需求使用 A Scaling] 一文。[57]

安装应用程序代码

当您配合 Elastic Beanstalk 使用 Auto Scaling 时，Elastic Beanstalk 会负责在向上规模扩展时为的 EC2 实例安装应用程序代码。这也正是 Elastic Beanstalk 所提供的容器托管优势之一。

然而，如果大家单纯使用 Auto Scaling 而未配合 Elastic Beanstalk，则需要自行在新 EC2 实例上装应用程序代码以实现自动规模伸缩。如果您选择了 Chef 或者 Puppet，可以考虑利用其在实例上部署应用程序代码。AWS OpsWorks Auto Scaling 亦可利用 Chef 配置实例，具体包括提供一Auto Scaling 变种以实现基于时间以及基于负载的自动规模伸缩能力。[58] 在 OpsWOrks 的帮助下您也可以为自己的实例设置定制化启动与关闭条件。OpsWorks 是一套出色的自动规模伸缩备选案，非常适合已经使用 Chef 或者有意使用 Chef 管理 AWS 资源的用户。欲了解更多细节信息，请阅 AWS OpsWorks 用户指南中的利用基于时间及基于负载实例管理负载部分。 [59]

如果您不打算使用上述软件包，也可以直接使用 Ubuntu cloud-init 软件包作为最简单的方法，从将 shell 命令传递至 EC2 实例处 [60]。您可以使用 cloud-init 运行一套简单的 shell 脚本以提取最新用程序代码并启动与之对应的服务。Amazon Linux AMI 官方支持这类操作，Canonical Ubuntu A同样提供官方支持 [61]。欲了解更多细节信息，请参阅 AWS 架构中心页面。[62]

游戏服务器

游戏服务器的实现方法与 RESTful 正好相反。其中客户端建立一条指向游戏服务器的有状态双工接，具体可通过 UDP、TCP 或者 WebSockets 实现，从而使客户端与服务器能够彼此实现消息发起如果该网络连接中断，则客户端必须执行重连逻辑，甚至有可能需要重置逻辑的状态。有状态游服务器之所以会给自动规模伸缩带来挑战，是因为在这类情况下，客户端无法直接跨越服务器池行负载均衡循环。

从历史角度看，很多游戏都会使用有状态连接并配合长期运行服务器进程以实现自身全部功能，别是在 3A 级大作以及 MMO 游戏当中。如果您的游戏属于这类架构形式，则可将其运行在 AWS上。我们在 Amazon Gamelift 当中提供一项托管服务，能够帮助您轻松完成基于会话的多人玩家戏的部署、运营以及专用游戏服务器规模伸缩。另外，您也可以选择通过 Amazon EC2 运行自己游戏服务器编排机制。这两种选项皆行之有效，具体取决于您的实际需求。不过对于新游戏而言我们建议大家尽可能使用 HTTP，且仅在必要时（例如多人在线游戏）使用有状态套接。

以下表格列出了您可用于构建事件驱动型服务器的几款软件包。

语言	软件包
Ruby	Event Machine[63]
Python	gevent,[64] Twisted[65]
Node.js	Core,[66] Socket.io,[67] Async.js[68]
Erlang	Core[69]
Java	JBoss Netty[70]

表中之所以未列出 C++，是因为其几乎已经成为多人游戏服务器的首选语言。包括 Amazon Lumberyard 以及虚幻引擎在内的大多数游戏引擎都使用 C++编写而成。不过 Unity 作为少数例外则由 C#编写而成。这意味着您可以从客户端中提取现有游戏代码，并在服务器上加以复用。这种方法对于运行物理服务器或者在服务器上运行其它框架（例如 Havok）的情况极具价值，而这些方通常仅支持 C++。不过这里需要强调，您所选择的服务器编程语言并不重要。网络与数据库延迟是游戏服务器所面临的最大局限性因素。大家应当选择自己最为熟悉的语言选项，从而确保尽可降低编程、监控与调试难度。

无论您选择了哪一款编程语言，有状态套接服务器都能够从可观的实例规模中获益，这是因为此服务器对于网络延迟等问题更为敏感。在 Amazon EC2 计算优化实例家族（例如 c4.*）当中，规最大的实例往往就是佻的最佳选项。这些新生代实例通过单 root I/O 虚拟化（简称 SR-IOV）显增强网络传输能力，从而实现更高每秒数据包传输、更低延迟以及少卡顿效果——这一切都令其为游戏服务器的理想选择。欲了解更多细节信息，请参阅 AWS 白皮书《在 AWS 上优化多人游戏务器性能》。[71]

玩家匹配

玩家匹配是吸引玩家加入游戏的一类功能。一般来讲，玩家匹配的流程将如下所示：

1. 询问用户其希望加入的游戏类型（例如死亡竞赛、时间挑战等等）。
2. 查看目前线上正在进行的游戏模式。
3. 各类变量因子，包括用户地理位置（以核算延迟）或者 ping 时长、语言以及整体排名等
4. 将用户放置在包含玩家匹配游戏的游戏服务器上。

游戏服务器需要长期运行的进程，其无法像 HTTP 请求那样单纯通过负载均衡循环进行负载分发
在游戏玩家被分配至某一服务器时，其在游戏结束之前将始终处于该服务器之上，且具体时长可
从数分钟到数小时不等。

在现代云架构当中，大家应当尽可能控制长期运行游戏服务器进程的使用率，即保证仅保留必要
游戏元素。举例来说，我们假定存在一款 MMO 或者开放世界射击游戏。其中部分功能——例如
世界各地探险并与其他玩家互动——需要长时间运行的游戏服务器进程。然而，其它 API 操作—
例如罗列好友名单、变更库存、更新统计数字以及寻找竞赛——皆可通过直接映射至 REST Web A
来实现。

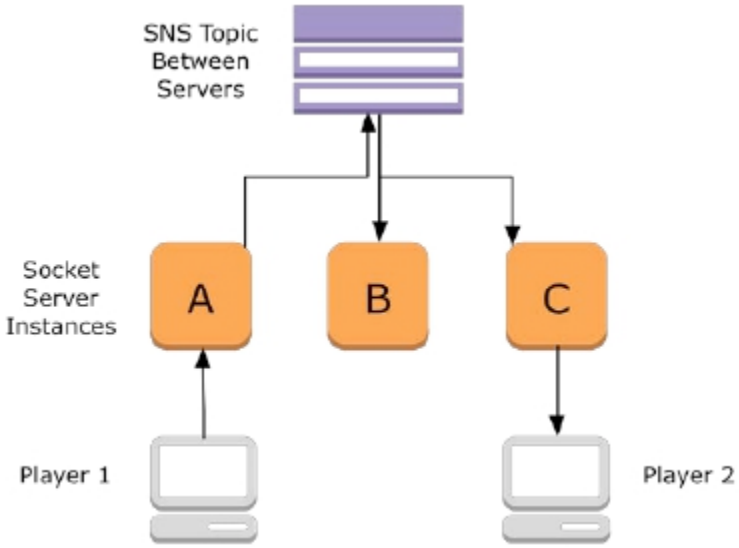
在这套方案当中，游戏客户端会首先连接到您的 REST API，并请求有状态游戏服务器。您的 RE
API 随后会执行匹配逻辑，并为客户端提供一条 IP 地址与服务器端口以供接入。该游戏客户端随
会直接连入该游戏服务器的 IP 地址。

这种混合方案能够使您的套接服务器获得最佳性能，因为客户端将能够直接接入 EC2 实例。与此
时，您仍然可以享受在主入口点利用基于 HTTP 的调用所带来的种种助益。欲了解更多在定制化
服务器环境下实现匹配功能的细节信息，请参阅 Lumberyard & Amazon Gamelift 博客当中的《
合模式：利用 Amazon GameLift 实现无服务器定制化匹配》一文。[72]

利用 Amazon SNS 推送消息

在游戏当中进行消息推送的方法主要分为两大类：面向特定用户的消息——例如游戏邀请或者移
推送通知; 以及组消息，例如聊天或游戏数据包。在发送与接收此类消息时，最常见的实现方式是
用一套配备有状态连接的套接服务器。如果您的玩家群体规模较小，即每位玩家皆可接入同一台
务器，那么大家可以通过选择不同套接的方式在玩家间进行消息路由。不过在大多数情况下，您
能需要使用多台服务器，这意味着这些服务器同样需要通过某种方式实现彼此之间的消息路由。

要在各 EC2 服务器实例之间进行消息路由，Amazon SNS 能够帮上大忙。我们假定您有一位玩家
在服务器 A 上，其希望向身处服务器 C 的玩家 2 发送一条消息，具体如下图所示。在这样的情况下
服务器 A 会查找本地接入玩家，并在无法找到玩家 2 时将消息转发至一个 SNS 主题——后者随后
将消息传播至其它服务器处。



图三：SNS 支持下的服务器间玩家到玩家通信机制

Amazon SNS 在这里的作用类似于 RabbitMQ 或者 Apache ActiveMQ 等消息队列方案。当然，大
也可以在 Amazon EC2 实例上直接运行 RabbitMQ、Apache ActiveMQ 或者其它类似软件包。不
Amazon SQS 的优势在于，大家不必投入时间与精力来管理并维护自己的队列服务器与软件。欲
解更多与 Amazon SNS 相关的细节信息，请参阅 Amazon SNS 开发者指南当中的 Amazon 简单通
服务是什么？[73] 以及创建一个主题 [74] 等内容。

移动推送通知

Amazon SNS 还支持面向移动客户端直接发送推送通知的能力。与之前提到的专门负责处理近实
游戏内消息收发的用例不同，移动推送更适合在玩家退出游戏时向其发送消息。具体实例主要为
户特定类事件，例如某位好友超过了您的最高得分，或者双倍经验活动即将开启等。

移动推送属于 Amazon SNS 的一项延伸功能，因此其继续沿用相同的主题概念，但配合另一不同
点。您的服务器代码会根据某些游戏内事件向合适的 SNS 主题内添加消息，而后该消息会被交付
对应用户的设备端。Amazon SNS 移动推送机制能够处理不同推送通知平台之间的 API 差异，具
包括 APNS、GCM 以及 ADM 等等。欲了解更多相关信息以及完整的受支持平台列表，请参阅 Amaz
SNS 开发者指南当中的 Amazon SNS 移动推送通知内容。[75]

总结思考

相信很多朋友都执着于找到最完美的编程框架或者模式。RESTful 与有状态游戏服务器都拥有自
的优势，而之前提到过的各类编程语言也都能够在合理运用之下发挥卓越效果。更重要的是，您
要投入时间以思考整体游戏数据架构——包括数据存放在何时处、如何加以查询以及如何高效进
更新等等。

关系对 NoSQL 数据库

横向扩展型应用程序的出现已经彻底改变了应用层，同时颠覆了实现单一大型关系数据库的传统
法。目前，多种新兴数据库凭借着回避传统原子性、一致性、隔离性与持久性（简称 ACID）原则
同时最大程度强调轻量化接入、分布式存储以及最终一致性保证的特点而广受青睐。此类 NoSQL
数据库同样适用于游戏场景，其中的数据结构更多以清单以及集合的方式呈现（例如好友、排行榜
关卡、物品等），而非复杂的关系数据。

作为一项一般性规则，在线游戏最为核心的性能瓶颈往往源自数据库。一款典型的 Web 应用往往带来规模可观的读取操作与数量极低的写入操作——阅读博客、观看视频等都属于这种情况。但戏则恰恰相反，由于游戏当中的状态会持续变化，因此指向数据库的读取与写入操作也都一直非频繁。

目前关系数据库与 NoSQL 数据库领域都提供大量方案选项，不过在 AWS 上的游戏中使用频度最的当数 MySQL、Amazon Aurora、Redis、MongoDB 以及 Amazon DynamoDB。

首先，我们将从 MySQL 聊起——这是因为其非常适合游戏使用，且具备极高人气。另外，将 MyS与 Redis 相结合，或者将 MySQL 与 DynamoDB 相结合的作法在 AWS 上同样获得了成功。这里到的所有数据库备选方案皆支持对游戏而言至关重要的原子操作——例如递增及递减。

MySQL

作为一套 ACID 兼容型关系数据库，MySQL 拥有以下优势：

- **事务 –** MySQL 能够将多项变更以分线方式纳入单一原子事务之内，而后进行提交或者回滚

NoSQL 存储则通常缺少这种多步事务功能。

- **高级查询 –** 由于 MySQL 使用 SQL 语言，因此我们能够更为灵活地根据需求执行复杂查NoSQL 数据库通常仅支持按键或者按单一辅助索引形式访问。这意味着大家必须在初期即制定出密的数据设计思路。

- **单一真实来源 –** MySQL 能够保证内部数据一致性。而 NoSQL 类解决方案之所以通常速度快，则归功于其分布式存储与最终一致性机制。（最终一致性意味着您可以在一个节点上写入一个在另一节点上提取该键，但该键不会在另一节点上马上出现。）

- **广泛的工具 –** MySQL 自上世纪九十年代起就已经诞生，目前市面上存在大量与之相关的

试与数据分析工具。另外，SQL 还是一种易于理解的通用型语言。

上述优势的存在令 MySQL 保持着强大的吸引力，特别是在帐户记录、应用内购等对于事务及数
一致性要求极高的游戏功能方面。事实上，即使是已经开始大范围使用 Redis 以及 DynamoDB
NoSQL 方案的游戏开发商，也仍然在利用 MySQL 打理自己的帐户与购买等交易数据。

如果您正在 AWS 之上使用 MySQL，我们建议您使用 Amazon RDS 进行 MySQL 托管，这将帮助
家节约宝贵的部署与支持周期。Amazon RDS for MySQL 能够自动完成各类极耗时间的数据库管
任务，具体包括启动 EC2 实例、配置 MySQL、附加 Amazon 弹性块存储（Amazon Elastic Block Sto
简称 EBS）分卷、设置复制、运行夜间备份等等。另外，Amazon RDS 还提供多种先进功能，包
多可用区间同步复制以实现高可用性、自动化主-从故障转移以及读取副本以提升性能表现等。要
解如何使用 Amazon RDS，请参阅启动 MySQL 数据库实例一文。[76]

以下是几项我们建议您在创建 RDS MySQL 数据库实例时采用的配置选项：

- 数据库实例类型：在开发/测试环境中使用 Micro 实例，在生产环境下使用 Medium 或者
大的实例。

- 多可用区部署？对于开发/测试环境，并无此必要；但对于生产环境，请记得启用同步多可
区复制与故障转移。要获得最佳性能表现，请始终将生产负载所在的 RDS 数据库实例同其它 Amaz
RDS 开发/测试数据库实例隔离开来。

- 自动次级版本升级？是的，这将帮助您有效削减日常维护工作量。

- 分配存储资源：开发/测试环境下为 5 GB，生产环境下则最少需要 100 GB 以支持配置 IOP

- 是否使用配置 IOPS？在生产环境下答案为是。配置 IOPS 能够确保提供特定级别的磁盘性
这一点对于大规模写入负载而言非常重要。欲了解更多与配置 IOPS 相关的细节信息，请参阅
Amazon RDS 用户指南当中的利用 Amazon RDS 配置 IOPS 存储改进性能表现。[77]

- 在玩家数量较低的时段内——例如明天清晨——进行 Amazon RDS 备份快照与升级操作。
果可以，尽量避免在这一时间窗口内运行后台作业或者夜间报告，从而防止查询积压问题。

- 要找到并分析生产环境下的缓慢 SQL 查询，您需要确保在 Amazon RDS 当中启用 MySQ
慢查询记录——具体如下所示。这些设置需要利用 Amazon RDS 数据库参数组 [78] 进行配置。需要
意的是，进行慢查询记录可能带来稍许性能损失。

  - 设置 `slow_query_log = 1` 以启用。在 Amazon RDS 当中，慢查询会被写入 `mysql.slow_log` 表当中。

  - `long_query_time` 当中的设定值用于确定仅计入时长超过特定秒数的查询。其默认
定值为 10，您可以根据实际情况将其设定为 5、3 甚至是 1。

  - 确保定期轮换慢查询记录，具体请参阅 Amazon RDS 用户指南当中的 MySQL 数据库
例常规数据库管理任务。[79]

随着游戏规模的不断扩大以及写入负载的持续增长，您需要重新调整 RDS 数据库实例以进行向上
展。对 RDS 数据库实例进行规模调整会带来停机时间，但如果您将其部署在多可用区模式之下，
能够保证通过故障转移机制控制生产环境的停机时长（通常为数分钟）。欲了解更多细节信息，请
阅 Amazon RDS 用户指南中的对运行 MySQL 数据库引擎的数据库实例加以修改一文 [80]。另外，
也可以添加一套或者多套 Amazon RDS 读取副本，用以分担主 RDS 实例所面对的读取负载强度
确保其始终拥有充足的资源处理数据库写入周期。欲了解更多与 Amazon RDS 副本部署相关的细
信息，请参阅如何使用读取副本一文。[81]

Amazon Aurora

Amazon Aurora 是一款 MySQL 兼容式关系数据库引擎，其将高端商用数据库的速度与可用性优
同开源数据库的简易性与成本效益优势结合在一起。Amazon Aurora 能够为游戏工作负载带来以
重要助益：

- 高性能 – Amazon Aurora 在设计层面能够立足同样的底层硬件提供五倍于标准 MySQL 的
据吞吐量。这样的性能水平堪比商用数据库，但其使用成本则明显更低。在规模最大的 Amazon
Aurora 实例当中，用户每秒最多可进行 50 万次读取与 10 万次写入，且各读取副本间的延迟在
毫秒左右。

- **数据持久性** – 在 Amazon Aurora 当中，您的数据库分卷将以 10 GB 为分块单位跨越 3 个用区进行六向复制，这意味着其中两套数据副本丢失将不会对数据库的写入可用性造成任何影响而三套数据副本丢失不会影响数据库的读取可用性。备份以自动化方式持续保存在 Amazon S3 后者拥有高达 99.999999999%设计持久性，保留周期为 35 天。您可以在保留周期之内随时对数库进行恢复，且最短间距为 5 分钟之前。

- **可扩展性** – Amazon Aurora 能够自动将存储子系统的容量扩展至最高 64 TB。这部分存储源将自动进行配置，因此您不必提前为此费心。而由此带来的另一项助益在于，您只需要为自己际使用的资源付费，这将显著降低规模扩展成本。Amazon Aurora 还能够以任意可用区组合方式署最多 15 套读取副本，甚至允许您跨越已上线 Amazon Aurora 的多个服务区。通过这种方式，将能够在遭遇实例故障时实现无缝化故障转移。

如果您利用 Amazon Aurora 支持游戏工作负载，请参考以下几项相关建议：

- 使用以下数据库实例类型：t2.small 实例适用于开发/测试环境，而 r3.large 或者规模更大实例则适用于生产环境。

- 至少在一个额外可用区内部署读取副本，从而实现故障转移与读取操作负载分摊效果。

- 在玩家在线数量较低时进行 Amazon RDS 备份快照以及升级操作。如果可能，请尽可能避在此时间窗口内运行作业或者与数据库相关的报告任务，否则可能造成积压问题。

如果您的游戏规模已经超出传统关系数据库——例如 MySQL 或者 Amazon Aurora——的承受能我们建议您首先执行性能评估，包括调整参数以及拆分。另外，您也应审视是否存在其它适用的 NoSQL 类产品——例如 Redis 或者 DynamoDB，尝试利用其分担一部分 MySQL 工作量。在接下的章节中，我们将专门介绍几种高人气 NoSQL 解决方案。

Redis

作为一款原子数据结构服务器方案，Redis 具备一系列其它数据库不具备的独特优势。Redis 提供
种基础数据类型，包括 counter、list、set 以及 hash，且通过一项基于文本的高速协议实现访问
欲了解更多与 Redis 数据类型相关的细节信息，请参阅 Redis 数据类型说明文档[82] 与 15 分钟 Re
快速指南[83]。对这些独特数据类型的支持能力，使得 Redis 成为排行榜、游戏列表、玩家计数、
态、库存以及其它类似数据的良好解决方案。Redis 将整体数据集放置在内存当中，因此能够提
极快的访问速度。欲了解 Redis 与 Memcached 之间的对比差异，请参阅 Redis 基准测试。[84]

不过这里还要强调您应当了解的几条 Redis 相关注意事项。首先，您需要为其准备大量物理内存
源，因为其整体数据集都将驻留在内存当中（且不支持虚拟内存机制）。另外，Reids 的复制支持
为简单化，调试工具选项也比较有限。具体来讲，Redis 不适合作为惟一的数据存储方案使用。
如果配合磁盘支持型数据库——例如 MySQL 或者 MongoDB，则 Redis 将能够为游戏数据提供出
的可扩展性提升。Redis 加 MySQL 亦是目前大受欢迎的游戏类解决方案。

Redis 对 CPU 资源的需求量不高，但却会占用大量内存资源。因此，我们最好将其与高内存实例
行配伍，例如 Amazon EC2 内存优化型实例家族（即 r3.*）。AWS 提供一项全面托管的 Redis 服务
即 Amazon ElastiCache for Redis[85]。ElastiCache for Redis 可处理集群化、主-从复制、备份以及
它各类常见的 Redis 维护任务。欲了解更多与 ElastiCache 相关的细节信息，请参阅 AWS 白皮书
用 Amazon ElastiCache 实现规模化性能保障》。[86]

MongoDB

MongoDB 是一套面向文档的数据库方案，这意味着其中的数据被存储为嵌套式数据结构——类
于典型编程语言中所使用的结构。MongoDB 还使用 JSON 的一套二进制版本——即 BSON 进行
信，如此一来我们即可以存储及检索 JSON 结构的方式对其进行编程。由于服务器 API 通常也采
JSON 格式，因此上述特性使得 MongoDB 在游戏与 Web 应用领域受到热烈欢迎。

MongoDB 还提供其它一些有趣的混合特性，包括使用类 SQL 式语法以允许用户通过范围及复合
件进行数据查询。

MongoDB 支持各类原子操作，例如递增/递减以及面向列表的添加/移除; 这一点与 Redis 比较相
欲了解 MongoDB 所能够支持的原子操作示例，请参阅 findAndModify 上的 MongoDB 说明文档
以及 MongoDB Cookbook[88]。

MongoDB 在游戏领域被广泛用于一级数据存储，且通常与 Redis 配合使用，从而发挥二者间的互
效果。所有临时性游戏数据、会话、排行榜以及计数器皆被保存在 Redis 当中，而后将进度以逻
点的形式存储于 MongoDB 内（例如关卡结束时或者新成就解锁时）。Redis 能够为延迟敏感型游
数据提供高速访问支持，而 MongoDB 则负责简化数据持久性的实现流程。

MongoDB 还支持原生复制与拆分机制，不过您必须亲自对这些功能进行配置与监控。欲了解更
在 AWS 之上部署 MongoDB 的细节信息，请参阅 AWS 白皮书《在 AWS 上运行 MongoDB》。[8

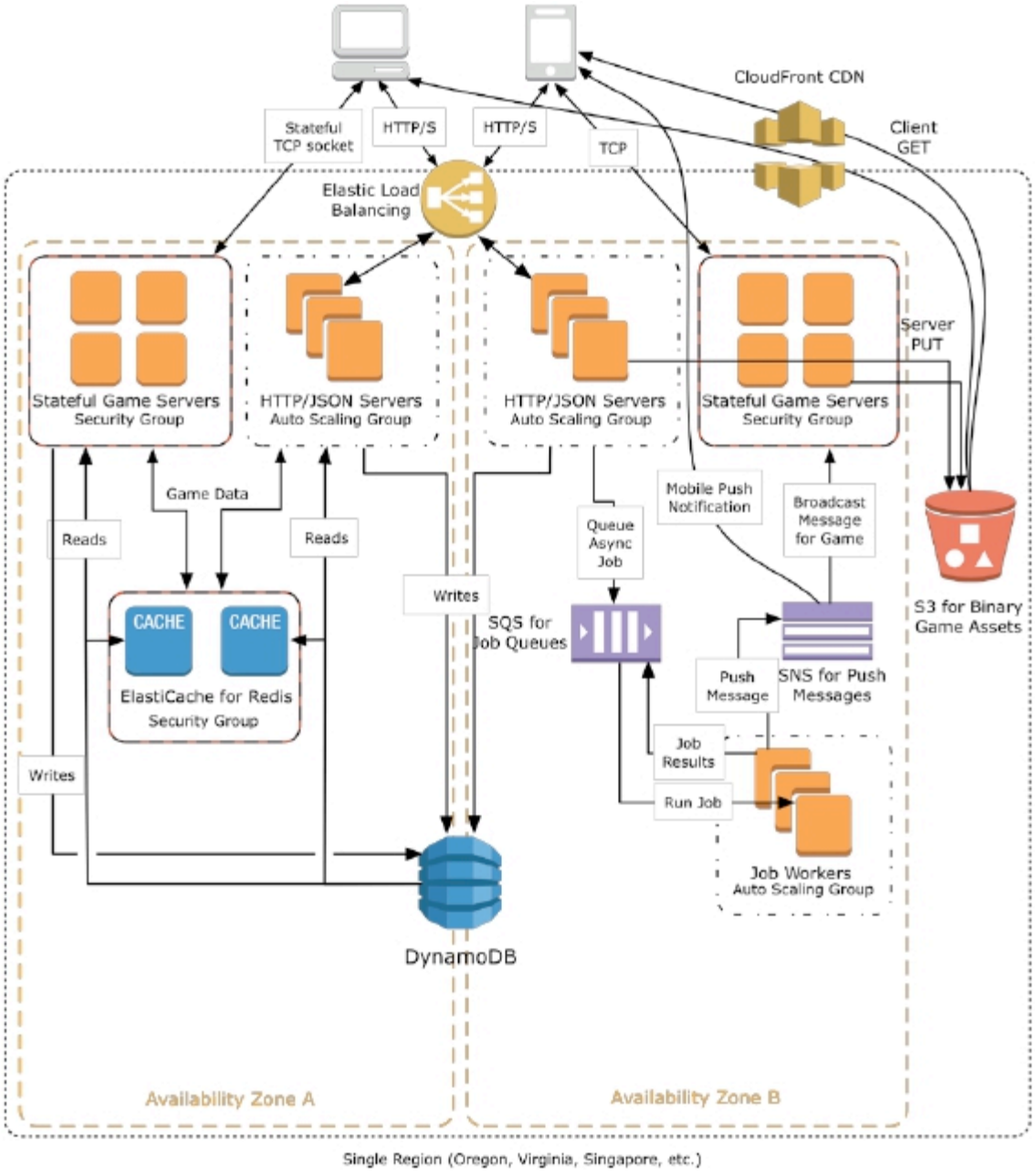
Amazon DynamoDB

最后，DynamoDB 是一套由 AWS 提供的全面托管 NoSQL 解决方案。DynamoDB 能够管理多种
见任务，包括同步复制以及 IO 配置等等，同时自动实现规模伸缩与缓存管理。DynamoDB 采用
套预配置数据吞吐量模式，您可以在其中指定每秒需要进行多少次读取与写入操作，其余部分
DynamoDB 会为您自动调节完成[90]。DynamoDB 的相关支持库提供多种语言版本，具体包括 Ru
AWS SDK[91]、Java AWS SDK[92] 以及支持 AWS 的 Python Boto 库。[93]

要了解如何设置 DynamoDB，请参阅相关上手指南[94]。一般来讲，游戏会频繁通过以下方式使用
DynamoDB 功能：

- 面向用户数据、物品、好友及历史的键-值存储。
- 面向排行榜、分数及面向日期类数据的范围键存储。
- 面向游戏状态、用户数量以及玩家匹配的原子计数器。

与 MongoDB 以及 MySQL 类似，DynamoDB 也能够与 Redis 等其它技术方案进行配伍，从而处
实时排序与原子操作等任务。多数游戏开发者发现，DynamoDB 自身已经足以满足其实际需求。
添加 Redis 或者为基于 DynamoDB 的架构引入缓存层的作法，确实能够令其灵活性更上一层楼。
面让我们重温之前提到过的参考示意图，观察 DynamoDB 如何在其中简化整体架构。



图四：利用 **DynamoDB** 运行在 **AWS** 之上的完整生产就绪游戏后台

表结构与查询

与 MongoDB 类似，DynamoDB 同样属于一套松散结构的 NoSQL 数据存储体系，其允许用户以一记录为基础保存不同属性集。您只需要对自己打算使用的主键策略进行预先定义即可：

- **分区键** – 分区键属于单一属性，DynamoDB 利用其作为内部散列函数的输入内容。分区键以是玩家名称、游戏 ID、UUID 或者其它类似的惟一键。Amazon DynamoDB 会为此键构建起一无序散列索引。

- **分区键与排序键** – 这类键被称为复合主键，其中包含两项属性：分区键与排序键。Dynamo利用其中的分区键值作为内部散列函数的输入内容，且全部具有该相同分区键的条目都将被一并储，而具体顺序则由排序键值决定。举例来说，您可以将游戏历史记录存储为[`user_id`, `last_login`]的配对形式。Amazon DynamoDB 会为分区键构建起一套无序散列索并根据排序键属性建立有序范围索引。在这种情况下，只有将两种键相结合方可指向惟一对象。

为了实现最佳查询性能，您应将各个 DynamoDB 表的大小维持在可管理的范围之内。举例来说，果您拥有多种游戏模式，则最好能够为每种游戏模式建立一套独立的排行榜表，而非将其全部集在一套大表当中。这种处理方式还能为您带来对其中特定排行榜进行单独规模扩展的灵活性——如其中某种游戏模式的人气远高于其它。

配置数据吞吐量

DynamoDB 会在幕后对您的数据进行拆分，从而满足您对于数据吞吐量的特定要求。DynamoD定义了读取与写入单元的概念。一个读取容量单元代表着一次强一致性每秒读取，或者两次最终致性每秒读取，其中条目最大不超过 4 KB。默认情况下的设置为 5 读取与 5 写入单元，这意味着秒强一致性读取数量量为 20 KB，而每秒写入数据量为 5 KB。您可以随时调整帐户限额以提升读与/或写入容量，也可以随时对两项指标作出下调——但请注意，每天的下调次数不可超过 4 次。可以使用 AWS 管理控制台或者 CLI 以选定对应表并进行修改，从而完成上述规模伸缩操作。另外您也可以使用 Auto Scaling 服务以动态方式根据需求调整预配置数据吞吐量，这种根据实际流量式作出响应的 DynamoDB Auto Scaling 组合拥有诸多优势 [95]。DynamoDB Auto Scaling 还能够Amazon CloudWatch 警报机制相结合，由后者负责对各容量单元加以监控。其会根据您定义的规进行规模调整。

新近配置完成的数据吞吐量需要经过一定延后方可正式起效，此时数据将在后台进行重新分区。不会造成任何停机时间，因此您可以充分利用 DynamoDB 的这种规模伸缩能力解决随时间推移产的需求变化——例如游戏玩家由 1000 名增长至 10000 名。不过 DynamoDB 在设计层面并不适处理以小时为周期的用户峰值状况。要解决这类需求，您应当选择其它数据库，同时利用某种形的缓存机制提升弹性水平。

要真正发挥 DynamoDB 的最佳性能，您需要确保自己的读取与写入操作尽可能均匀地散布于全部上。使用散列键或者 checksum 等十六进制字符串能够轻松实现这种随机性效果。欲了解更多与 DynamoDB 性能优化相关的细节信息，请参阅 Amazon DynamoDB 开发者指南中的 DynamoDB 佳实践 [96] 部分，以及 AWS 博客发布的在 Amazon DynamoDB 中优化配置数据吞吐量 [97]。

Amazon DynamoDB 加速器(简称 DAX)

DAX 允许您为 DynamoDB 提供一套全面托管的内存内缓存层，用以将 DynamoDB 表的响应速度毫秒级别改善至微秒级别。这种加速机制无需对您的游戏代码作出任何大幅改动，因此能够显著化其在您架构当中的部署流程。您需要做的仅仅是利用指向 DAX 的新端点对 DynamoDB 客户端行重新初始化，其余部分代码皆不会因此受到任何影响。DAX 会自行处理缓存失效及数据填充等务，无需用户为此劳心。该缓存方案能够快速应对各类可能引发玩家数量激增的活动事件，例如度 DLC 发布或者新补丁上线等等。

其它 NoSQL 选项

目前市面上还存在着其它多种 NoSQL 备选方案，具体包括 Riak、Couchbase 以及 Cassandra 等等您可以根据需求随意选择，而且也已经存在诸多成功案例表明其完全能够在 AWS 之上取得成功。如选择服务器编程语言一样，世界上并不存在一种完美的数据库选项——您需要权衡各类解决方的各自优势与短板。

缓存

对于游戏而言，在数据库之前为需要频繁使用的数据添加缓存层能够有效降低扩展性问题的数量
即使是面向排行榜、好友列表乃至近期活动的，存在周期仅为数秒的短期缓存，也足以帮助您的
据库分担大量负载。另外，添加缓存服务器在实现成本上也远低于添加额外数据库服务器，这显
有助于进一步降低 AWS 资源的使用开销。

Memcached 是一种高速、基于内存的键-值存储方案，并已经成为缓存领域的黄金标准 [98]。近年来
Redis 则逐渐凭借着与 Memcached 类似的性能表现外加先进的数据类型支持能力 [99] 而广受青睐
两种选项都能够在 AWS 之上顺利运作。您可以在 EC2 实例之上自行安装 Memcached 或者 Red
也可以利用 Amazon ElastiCache[100] 这项由 AWS 托管的缓存服务实现提速效果。与 Amazon RDS
DynamoDB 类似，ElastiCache 能够以完全自动化方式在 AWS 之上完成 Memcached 与 Redis 的
装、配置与管理。欲了解更多与 ElastiCache 设置相关的细节信息，请参阅 Amazon ElastiCache
户指南中的 Amazon ElastiCache 上手教程。[101]

ElastiCache 对集群中的服务器进行分组以简化管理流程。诸如配置、安全保护以及参数变更之类
ElastiCache 操作大多在缓存集群层加以执行。尽管使用集群技术，但 ElastiCache 各节点并不会
此通信或者共享缓存数据。ElastiCache 可根据您的选择部署 Memcache 及 Redis 版本，从而确保
Ruby、Java、PHP、Python 以及其它各类语言编写而成的现有客户端库能够与 ElastiCache 完全
容。

典型的缓存实现方案被称为惰性填充，或者预留缓存（cache aside）。这意味着缓存内容会接受检查
如果需要的值并不存在于缓存内（即缓存未命中），则检索该记录，将其存储在缓存内，而后返
以下 Python 示例即检查 ElastiCache 以查找某个值，并在缓存内不存在该值时查询数据库，而后
该值返回至 ElastiCache 以备后续查询。

```
cache = memcached.Client(["mycache...cache.amazonaws.com:11211"])
def get_user(user_id):
    record = cache.get(user_id)
    if record is None:
        # Run a DB query
        record = db.query("select * from users where id = ?",
user_id)
        cache.set(user_id, record)
    return record
```

惰性填充之所以成为最常见的缓存策略，是因为其仅向缓存当中填充客户端实际请求的数据。这
方式避免将很少（或者从未）使用的记录写入至缓存当中，或者在实际读取之前变更缓存内容。
前这种模式已经得到广泛使用，且普遍存在于各类主流 Web 开发框架当中，具体包括 Rails、Djan
以及 Grails 等等。但这种策略亦拥有自己的缺点，即当数据发生变化时，客户端发出的下一次请
一定会引发缓存未命中，这意味着新记录必须经历自数据库处查询并填充至缓存中这一过程。
基于这样的缺陷，第二种高人气缓存策略应运而生。对于您确信需要进行频繁访问的数据，我们
以直接将其填充至缓存当中以避免不必要的缓存未命中问题。在这种情况下，大家可以在更新记
时完成缓存填充，而不必等待客户端进行实际查询。但需要强调的是，如果您的数据始终处于快
变更状态，则其可能导致不必要的过高缓存写入量。此外，面向数据库的写入操作可能遭遇用户
验变慢，这是因为缓存自身也需要同时进行更新。

```
cache = memcached.Client(["mycache...cache.amazonaws.com:11211"])
def update_username(record):
    # Run a DB query
    record = db.query("update users set name = record.user_name
where id = ?", record.user_id)

    cache.set(record.user_id, record)

    return record
```

要在这两种策略当中作出选择，您首先需要确定自己的数据如何变更以及如何接受查询。

最后一种主流缓存选项为定时刷新。这种方式适用于提供跨越多条不同记录的数据资料，例如排
榜或者好友清单。在这种策略当中，您需要建立一项用于查询数据库的后台任务，且每隔几分钟
缓存进行一次刷新。这能够降低缓存面对的写入负载压力，同时因页面稳定周期更长而能够对更
上游数据进行缓存（例如 CDN 层）。

Amazon ElastiCache 规模伸缩

ElastiCache 能够简化您对缓存实例进行规模伸缩的流程。ElastiCache 能够在无需额外费用的前
下，在 CloudWatch 当中允许您访问多项 Memcached 指标。欲了解具体指标清单，请参阅 Amaz
ElastiCache 用户指南中的利用 CloudWatch 指标监控使用情况部分 [102]。您应根据这些指标设置
CloudWatch 警报，从而在发生缓存性能问题时及时作出提醒 [103]。您可以对这些警报进行配置，
括在缓存内存即将被占满时发送邮件，或者在缓存节点响应缓慢时发出提醒。我们建议您重点监
以下指标：

- **CPUUtilization** – Memcached 或者 Redis 所使用的实际 CPU 资源量。过高的 CPU 使用
通常代表存在问题。
- **Evictions** – 因内存空间不足而被迫清出的键量。这一指标应当为零。如果实际结果并非
近于零，您可能需要使用规模更大的 ElastiCache 实例。
- **GetHits/CacheHits 与 GetMisses/CacheMisses** – 缓存能够正确提供所请求键的频
命中百分比越高，则代表能够为您的数据库分担更多负载压力。
- **CurrConnections** – 目前已接入客户端的数量（具体取决于应用程序）。

总体而言，命中率、未命中率以及被迫清出这几项指标对于绝大多数应用程序都非常重要。如果
中率与未命中率间的比值过低，您应当检查应用程序代码以确保您的缓存代码与预期相符。正如
前所提到，正常的被迫清出量应当始终趋近于零。如果结果非零，那么您应当向上扩展 ElastiCa
节点以提供更多内存容量，或者检查缓存策略以确保仅缓存必要内容。欲了解更多参数与监控管
相关信息，请参阅 Amazon ElastiCache 用户指南中的参数与参数组 [104] 以及我该监控哪些指标？
内容。

另外，您也可以配置自己的缓存节点集群以跨越多个可用区为游戏缓存层提供高可用性。这种作
能够确保当某一可用区出现不可用状况时，您的数据库仍能够从容应对突如其来的请求峰值。在
建缓存集群或者向现有集群添加节点时，您可以为这些新节点选择具体可用区。您可以指定每个
用区内的具体节点数量，或者选择将节点分布在多个可用区中。

利用 Amazon ElastiCache for Redis，您可以在其它可用区内创建一套读取副本。当主节点发生故
时，AWS 会配置一个新的主节点。而如果无法配置新的主节点，您则可以选择将哪套读取副本提
为新的主节点。

ElastiCache for Redis 还能够在配合 Redis Engines 3 或者更高版本的情况下支持拆分式集群。您
以使用最多 15 个分区创建集群，并将整体内存内数据存储量提升至 3.5 TiB 以上。其中每个分区
多可拥有 5 套读取副本，这将允许您每秒处理 2000 万次读取与 450 万次写入操作。

将拆分模式与读取副本相结合，我们将能够显著提升整体性能与可用性水平。数据会跨越多个节
存储，而一旦主节点发生问题，各读取副本将支持起快速且全面自动化的故障转移效果。

要发挥拆分模式的各项优势，大家必须使用具备集群识别能力的 Redis 客户端。该客户端将把整
集群视为一套散列表，将总计 16384 个插槽均匀分布在这些分区中，而后将输入键映射至正确的
区处。ElastiCache for Redis 则将整体集群视为一个单元，从而实现备份与恢复能力。如此一来，
将不必着眼于单一分区进行考量或者备份管理。

利用 Amazon S3 存储二进制游戏内容

您的数据库负责存储用户数据，具体包括帐户、状态、物品、购买记录等等。不过对于同游戏相
的二进制数据，Amazon S3 则更为合适 [106]。Amazon S3 提供一个简单的基于 HTTP API，用以 P
（上传）及 GET（下载）文件。利用 Amazon S3，您只需要为实际存储及传输的数据量付费。Amaz
S3 的具体使用方法包括创建一个用于存储数据的存储桶，而后面向该存储桶进行 HTTP 请求往
欲了解更多与这一流程相关的细节信息，请参阅 Amazon S3 上手指南中的创建存储桶部分。[107]

Amazon S3 非常适合处理各类游戏用例，具体包括：

- **内容下载** – 包括游戏资产、地图、补丁以及 beta 测试数据等。
- **用户生成的文件** – 包括图片、头像、用户创建的关卡以及设备备份等。
- **分析** – 存储各类指标、设备日志以及使用模式等。
- **云存档** – 游戏存档数据以及设备间同步。

虽然大家可以在理论上将此类数据存储在数据库当中，但利用 Amazon S3 打理这部分数据能够带
以下几项突出优势：

- 将二进制数据存储在数据库内会占用大量内存与磁盘空间，同时消耗宝贵的查询资源。
- 客户端能够利用简单的 HTTP/S GET 请求直接从 Amazon S3 处下载内容。
- Amazon S3 在设计上承诺全年 99.999999999%持久性与 99.99%可用性水平。
- Amazon S3 原生支持 ETag、验证以及签名 URL 等多种功能。
- Amazon S3 可接入 CloudFront CDN 以面向大量客户端进行内容分发。[108]

考虑到以上优势，下面我们一同了解 Amazon S3 中最适合游戏需求的特性。

内容交付与 Amazon CloudFront

可下载内容（简称 DLC）已经成为现代游戏吸引玩家长期驻留的有力武器，同时亦成为一种重要
收入来源。用户希望厂商能够在游戏发布后的数个月甚至数年之内持续发布新角色、关卡以及挑战
而对于 DLC 盈利策略而言，以高性价比方式快速交付更新内容就成了决定其成败的关键所在。

尽管游戏客户端本身通常经由特定平台的应用商店进行分发，但游戏新版本的推出则不太适合继沿用这种笨拙而费时的方式。各类促销或者限时活动显然应该由开发者自己负责打理，且保证其会对服务器基础设施的其它部分产生影响。

如果您需要面向大量客户端发布内容（例如游戏补丁、扩展或者 beta 测试版本等），我们建议您 Amzon S3 之前使用 Amazon CloudFront[109]。CloudFront 在全球范围内设有大量入网点（简称 PO 其能够显著提升下载性能表现。另外，您也可以利用区域 CloudFront 进一步优化成本构成。欲了 更多相关信息，请参阅 CloudFront 常见问题解答 [110]，特别是其中的“CloudFront 能够如何帮助 降低成本？”问题。

最后，如果您预计需要大量使用 CloudFront，请与我们的 CloudFront 销售团队进行联系。Amaz 将为高用量客户提供远低于按需计费标准的产品折扣。[111]

利用 ETag 简化版本控制

正如前文所提到，Amazon S3 支持 HTTP Etag 以及 If-None-Match HTTP 标头，这些为 Web 开 者所熟知的机制却往往被游戏开发者们所忽视 [112]。这些标头能够帮助大家面向 Amazon S3 中的部 内容发送请求，且其中包含您已经拥有之版本的 MD5 校验码。如果您已经拥有最新版本，则 Amaz S3 会发出 HTTP 304 Not Modified 响应，或者在您需要时发出 HTTP 200 外加文件数据。欲了 这一调用流程的说明内容，请参阅维基百科网站中的 HTTP Etag 典型使用方法页面。[113]

以这种方式使用 ETag 将能够进一步提升 CloudFront 的实际效果，这是因为 CloudFront 同样支 Amazon S3 ETag。欲了解更多细节信息，请参阅 Amazon CloudFront 开发者指南中的面向 Amaz S3 源的请求与响应活动。[114]

最后，大家还必须通过 CloudFront 的地理定位功能实现地理目标或者限制访问。Amazon CloudFront 会检测客户所在国家，并将国家代码转发至您的源服务器。如此一来，您的源服务器可根据客户的地理位置为其提供个性化显示内容。这类内容包括经过翻译的本地化 RPG 文本包旨在确保玩家能够切实享受游戏乐趣。

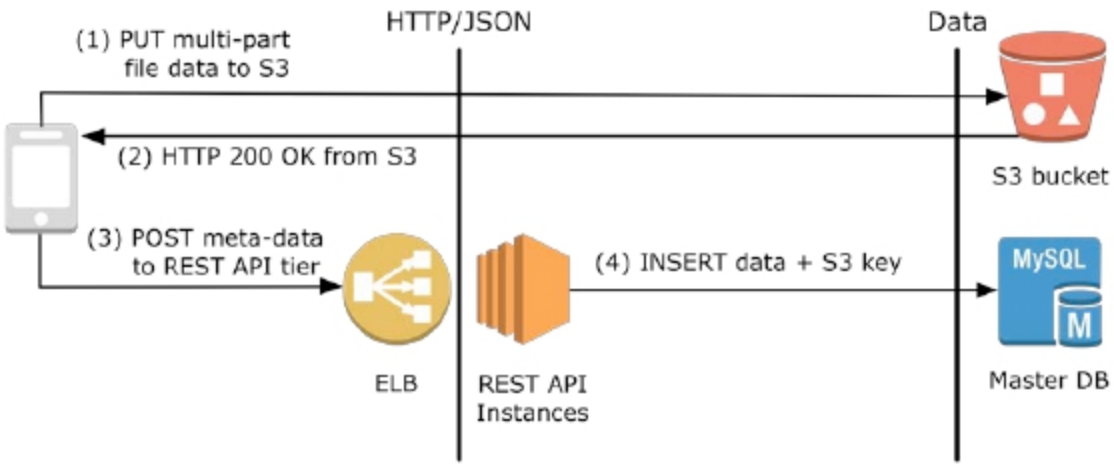
面向 Amazon S3 上传内容

对于其它一些游戏，我们可能需要面向 Amazon S3 上传游戏中的各类数据，包括用户生成的内分析或者游戏存档。我们可以通过两种方式完成 Amazon S3 数据上传：直接立足游戏客户端进行据上传，或者首先将其上传至 REST API 服务器，而后再由 REST 服务器将数据上传至 Amazon S虽然两种方法皆可起效，但我们建议大家尽可能直接面向 Amazon S3 进行上传，因为这有助于减您 REST API 层的负载强度。

直接向 Amazon S3 上传数据并非难事，甚至单纯通过网络浏览器即可完成。欲了解更多细节信请参阅 Amazon S3 开发者指南中的利用 POST 实现基于浏览器上传（AWS Signature v2）内容要保护数据免受损坏，您还应考虑对文件的 MD 校验码进行计算，并将其添加至内容 MD5 标头内如此一来，Amazon S3 将在上传过程中自动验证文件的完整性。欲了解更多细节信息，请参阅 Amazon S3 API 参考中的 PUT 对象部分。[116]

用户生成的内容（简称 UGC）属于面向 Amazon S3 上传的主要数据内容之一。典型的 UGC 主要为两部分：二进制内容（例如图形资产）以及元数据（例如名称、日期、作者、标签等）。常见的式是将二进制资产存储在 Amazon S3 当中，而后将元数据存储于数据库内。接下来，您即可利用数据库作为可用 UGC 的主索引，以供其他用户进行下载。

下图所示为一套调用流程，您可以借此将 UGC 上传至 Amazon S3 当中。



图五：游戏内容传输简单工作流

在以上示例中，我们首先将二进制游戏资产（例如头像、关卡等）PUT 至 Amazon S3 处，这将
Amazon S3 当中创建一个新的对象。在从 Amazon S3 处接收到成功响应后，您即可利用该资产的
数据向 REST API 层发出 POST 请求。该 REST API 需要具备一项服务以接收 Amazon S3 键名和
加您需要保留的一切元数据，而后将键名称与元数据存储在数据库内。在此之后，游戏的其它 RE
服务即可查询该数据库以获取新内容并进行下载。

这个简单的调用流程能够处理资产数据被存储在 Amazon S3 当中的用例——由用户创建的关卡或
角色通常会选择这种处理方式。同样的模式当然也适用于游戏存档——将游戏存档数据存储在
Amazon S3 当中，而后利用用户 ID、日期或者其它重要元数据在数据库内进行索引。如果大家需
对 Amazon S3 上传数据进行额外处理（例如生成预览图），请认真阅读本份白皮书后文中的异步
业章节。在该章节中，我们将探讨如何向队列任务内添加 Amazon SQS 以处理此类任务。

分析与 A/B 测试

收集与游戏相关的数据无疑是一项极为重要的任务，同时也是最易于完成的任务之一。其中最困
的部分可能反而在于判断到底该收集哪些内容。由于 Amazon S3 存储资源成本低廉，因此大家应
在无法明确判断指标重要性或者客户端难以更新的情况下，广泛收集与用户有关的各项相关指标
如总体游戏时长、最喜爱的角色或者物品，当前及最高等级等）。

然而，如果您能够明确找到希望回答的问题，或者客户端确实易于更新，则可专注于收集能够回
这些具体问题的相关数据。

在确定待收集数据之后，请按照以下步骤进行追踪：

1. 在用户设备（例如手机、游戏主机或 PC 等）上基于本地数据文件收集各项指标。为了简化
续处理流程，我们建议您使用 CSV 格式以及惟一文件名。举例来说，特定用户的数据可能被保存
241-game_name-user_id-YYYYMMDDHHMMSS.csv 或者其它类似的格式。

2. 由客户端将指标文件直接上传至 Amazon S3 以实现数据定期驻留。或者，您亦可选择集
Amazon Kinesis 以建立起一套我们之前提到过的松散耦合架构 [117]。在将特定数据文件上传至
Amazon S3 时，请使用新的文件名称打开一个新的本地文件，这样能够确保整个上传循环更为简
易行。

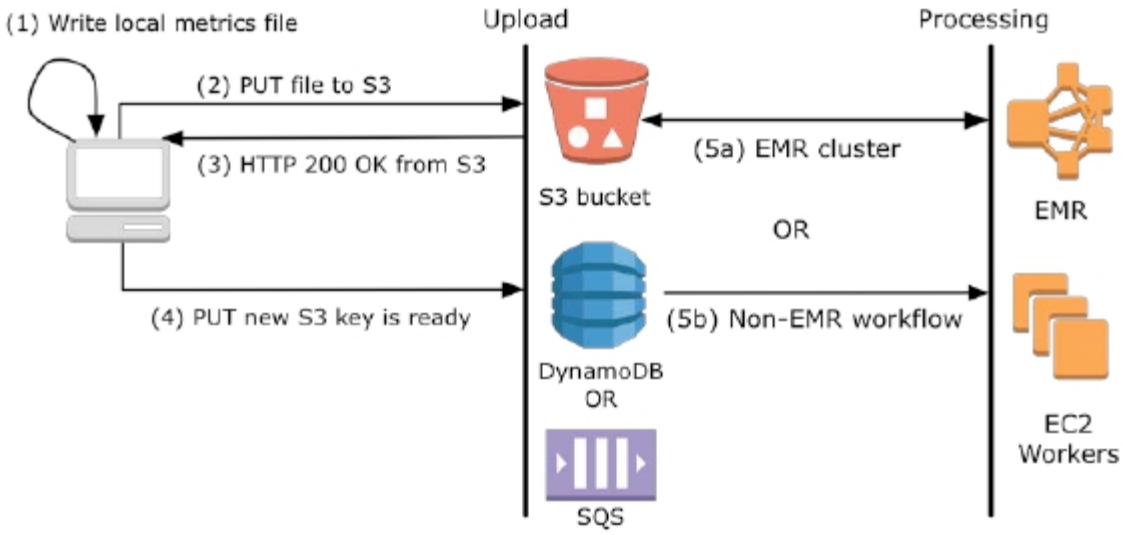
3. 对于您所上传的每一个文件，都应在记录中作出明确标识以提醒有新文件需要处理。Amaz
S3 事件通知机制能够很好地支持此类使用方式 [118]。要实现通知功能，您首先需要添加一条通知配
用以指明您希望 Aamzon S3 进行发布的事件类型（具体包括文件上传等）以及您希望 Amazon S
发送事件通知的目的地。我们建议使用 Amazon SQS，因为您随后可利用后台工作程序监控 Amaz
SQS 以发现新文件，并在其上传完毕后进行处理。欲了解更多细节信息，请参阅本份白皮书中的
Amazon SQS 章节。

4. 作为后台作业的组成部分，您还可以利用您选定并运行在 Amazon EC2 之上的 Amazon EM
或者其它框架进行数据处理 [119]。这一后台进程能够发现最后一次运行以来被上传的新数据文件，
时对该数据执行聚合或者其它操作。（请注意，如果您使用 Amazon EMR，则可能并不需要经历第
步，因为 Amazon EMR 内置对新文件流的支持能力。）

5. 作为可选方案，您可以将数据发送至 Amazon Redshift 当中以实现额外的数据仓储及分析
活性 [120]。Amazon Redshift 是一套 ANSI SQL 兼容性列式数据仓库方案，允许用户按小时计费使用
其能够帮助大家对大规模数据执行查询操作，包括使用您所熟悉的 SQL 兼容工具进行求和以及最
/最大值提取。

通过循环重复上述步骤，从而同步上传并处理数据。

以下示意图显示了这一模式的工作原理。



图六：简单分析与 A/B 测试流程

对于分析与 A/B 测试，其中的数据流趋于单向推进。具体来讲，指标流由用户处产生，接受处
而后由管理人员制定出可能影响未来内容发布或者游戏功能的决策。以 A/B 测试为例，当我们向
户呈现不同物品及屏幕内容等元素时，您可以记录用户作出的选择及其后续操作（例如购买、撤
等）。接下来，定期将此数据上传至 Amazon S3，并利用 Amazon EMR 创建报告。在最简单的用
之下，您可以直接从 Amazon EMR 处生成经过整理的 CSV 格式数据，并将其存储在其它 Amazon
存储桶当中，而后将其加载至电子表格程序之内。

本篇白皮书受篇幅所限无法对分析以及 Amazon EMR 作出深入探讨。欲了解更多信息，请参阅 A
大数据 [121] 以及 AWS 白皮书《Amazon EMR 最佳实践》[122]。若您需要与我们取得联系，请填写[面
游戏开发者的 AWS](#) 网站中的表单。

Amazon Athena

以快速且低成本的方式获取意见是开发者们改进自身游戏体验的重要方式之一。从传统角度讲，于此类数据往往必须从游戏应用服务器中提取、将其存储在某处、加以转化、加载至数据库内以查询，这样繁琐的流程使得获取意见变得相当困难。事实上，由于流程本身需要显著的时间成本计算资源投入，因此大多数开发者都很难承担这样的执行开销。

Amazon Athena 能够利用标准 SQL 对您存储在 Amazon S3 当中的数据进行查询，从而协助用户现这样的分析管道。由于 Athena 为无服务器服务，因此用户不需要承担任何基础设施配置或者管任务，且通常无需在开始查询之前对数据进行转换。然而，在利用 Athena 进行查询之前，亦有以几项与性能优化相关的事项需要加以考量：

- **利用 Athena 执行即时查询** – 由于 Athena 的计费方式为每 TB 扫描数据 5 美元起，因此未运行任何查询时，您并不需要支付任何费用。Athena 非常适合在需要快速从数据中提取意见性息时使用，且无需预先执行任何提取、转换与加载（简称 ETL）流程。
- **合理分区** – 数据分区能够将表拆分成多个部分，并保持各相关条目间的关联性。分区本身似于虚拟列。您可以在表创建时对其进行定义，而这些分区将帮助您降低每查询数据扫描量，从提升性能表现并降低特定查询的实现成本。您可以通过按查询、按特定过滤条件且基于特定分区方式限制数据扫描量。这里以下列查询为例：

```
SELECT count(*) FROM lineitem WHERE l_gamedate = '2017-02-01'
```

如果使用非分区表，则必须对整体表进行扫描，具体包括查看数百万条记录乃至数 GB 数据这将严重拖慢查询速度并带来不必要的成本。对表进行合理分区则有助于提高查询速度，并凭借 Athena 带来的查询数据量削减极大降低成本。欲参考具体示例，请参阅 AWS 大数据博客中的 Amazon Athena 十大性能调优提示一文。[1][2][3]

- **压缩** – 与分区类似，对数据进行合理压缩同样能够通过降低数据体积的方式改善网络负载成本。另外，我们最好确保所选择的压缩算法允许进行文件拆分，这样 Athena 的执行引擎就能够高并发方式进一步提升处理性能。
- 了解 **Presto** – 由于 Athena 使用 Presto——一套开源分布式 SQL 查询引擎——对从 GB 别到 PB 级别的不同规模数据源进行交互式分析查询，因此对 Presto 的透彻理解将帮助大家进一优化运行在 Athena 之上的各类查询。举例来说，ORDER BY 子句负责以排序顺序返回查询结果要进行排序，Presto 必须将全部数据行发送至单一工作程序处以进行排序操作。这显然会对 Pre 的内存资源产生巨大压力，甚至可能导致查询执行耗费太长时间。更糟糕的是，查询也许会因此失败。如果您使用 ORDER BY 子句查看顶部或底部的 N 个值，随后使用 LIMIT 子句将排序任务送至有限的几个工作程序处，则可带来远低于由单一工作程序负责执行的排序成本。

Amazon S3 性能注意事项

Amazon S3 能够通过扩展实现每秒数万次 PUT 与 GET 操作。为了实现这样的规模水平，我们需遵循几项指导原则以充分发挥 Amazon S3 的性能潜力。首先，在配合 DynamoDB 时，请确保您 Amazon S3 键名称为均匀分布——这是因为 Amazon S3 会基于键名称中的前几个字符进行内部据分区。

我们假定您的存储极名为 mygame-ugc，而您基于一条连续数据库 ID 进行文件存储：

```
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
```

在这种情况下，全部文件可能都将存放在同一内部分区当中，因为所有键名则以 107 作为开头。将极大限制您的可扩展能力，因为写入操作会以序列方式聚合在一起。比较简单的解决方案是使一项散列函数生成对象名称的开头部分，从而实现名称的随机分布。这里建议大家使用文件名的 MD5 或者 SHA1 哈希值作为开头，Amazon S3 键作为后续，具体如以下代码所示：

```
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
```

以下为 Python SHA1 版本的示例：

```
#!/usr/bin/env python
import hashlib
sha1 = hashlib.sha1(filename).hexdigest()[0:3]
path = sha1 + "-" + filename
```

欲了解更多提升 S3 性能表现的细节信息，请参阅 Amazon S3 开发者指南中的请求率与性能注意项部分 [124]。如果您预计将出现极高的 PUT 或者 GET 负载，请提交一份 AWS 支持申请，我们将此对您的存储桶架构进行针对性调整。[125]

松散耦合架构与异步作业

松散耦合架构的特征在于解耦组件，这是指在设计服务器组件时解除其关联关系以确保其尽可能独立形式运作。常见的实现方法是在各服务之间添加队列，这样一旦您系统当中的某一组件出现动峰值，也不会对其它组件造成影响。然而游戏当中的某些元素很难进行解耦，这是因为相关数需要及时更新以实现良好的匹配效果与游戏体验。然而，包括外观乃至角色数据在内的大部分数据并不需要实现毫秒级别的快速更新。

排行榜与头像

许多游戏任务都需要在后台中进行解耦与处理。举例来说，用户对自身状态的更新需要实时完成
这样如果某位用户已经在线并随后重新进入游戏，则其进度才不会丢失。不过在另一方面，我们
不需要在每当有用户获得新的高分时就对全球前百名排行榜进行更新。大多数用户并不会频繁查
排行榜。相反，我们可以将排名进程从评分发布中解耦出来，并在后台每几分钟执行一次。这几
不会对游戏体验造成任何影响，因为任何活跃的在线游戏都拥有快速变化的游戏排名结果。

这里再列举另一项实例，假定用户打算为自己的角色上传自定义头像。在这种情况下，您的前端
务器将该新头像的上传消息添加至 Amazon SQS 等队列机制当中。您需要编写一项定期运行的后
任务，其负责将头像从队列中提取出来、进行处理并确保其可供 MySQL、Aurora、DynamoDB
者您所选择的任何其它数据库使用。该后台作业运行在另一组 EC2 实例之上，且您可以根据需要
其设定为与前端服务器类似的自动规模伸缩形式。为了帮助您快速理清思路，Elastic Beanstalk
供多套工作程序环境，其将负责管理 Amazon SQS 队列并在各个需要从队列内读取数据的实例上
行守护进程，从而帮助您显著简化整个实现流程。[126]

这套方案能够有效将您的前端服务器从后台进程当中解耦出来，并帮助您对二者进行分别扩展。
例来说，如果头像图片的尺寸调整时间过长，您可以添加额外的作业实例，且无需调整您的 RE
服务器。

本章节的剩余部分将专注讨论 Amazon SQS。请注意，您也可以选择 RabbitMQ 或者 Apache ActiveMQ 等运行在 Amazon EC2 之上的其它备选方案。

Amazon SQS

Amazon SQS 为一套全托管队列解决方案，且具有一个长轮询 HTTP API[127]。这使其能够轻松与
所使用的各类服务器语言相对接。欲了解 Amazon SQS 的使用方法，请参阅 Amazon SQS 开发者
南中的 Amazon SQS 上手指南内容。[128]

- 在您 API 服务器所在的服务区内创建 SQS 队列，从而确保尽可能提升写入操作速度。您的
步工作程序则可处于任意服务区，因为其并不依赖时间因素。通过这种方式，您能够确保 API 服
器运行在距离您用户最近的服务区内，而作业实例则运行在更具经济效益的服务区中。

- Amazon SQS 在设计当中充分考虑到横向扩展需求。一个给定 Amazon SQS 客户端每秒能
处理约 50 条请求。您添加的 Amazon SQS 客户端进程越多，您能够并行处理的消息量就越大。
了解更多与工作进程添加与 EC2 实例相关的细节信息，请参阅 Amazon SQS 开发者指南中的利用
向扩展与批量机制提升数据吞吐量内容。[129]

- 您可以考虑利用 Amazon EC2 竞价实例作为作业工作程序，从而显著节约成本支出 [130]。
Amazon SQS 能够交付未被明确删除的消息，这将有效预防数据因 EC2 实例在作业过程中发生故
而遭遇消失。请确保仅在处理任务完成之后才将消息删除。如此一来，即使特定实例在运行当中
生问题，其它 EC2 实例也能够将其替代并重试同一项任务。

- 消息可见性同样与删除相关。基本上，我们可以将其视为未删除消息的重新交付时间，且
认为 30 秒。您可能需要延长这一时限以支持长期运行作业，从而避免多个队列读取程序接收到同
消息。

- Amazon SQS 还支持死信队列。所谓死信队列，是指其它（源）队列能够将其指定为消息
送目标，但却无法成功实现消息处理（消费）的队列。您可以预留并隔离死信队列中的消息，借
检查其处理流程为何无法成功。

另外，您还需要留心以下 Amazon SQS 注意事项：

- 并不保证消息能够按序送达。您可能会以随机顺序收取到消息（例如 2、3、5、1、7、6、
8）。如果您需要严格控制消息顺序，请参阅后文章节提到的 FIFO 队列。

- 消息通常能够快速收取，但偶尔也可能出现数分钟延迟。

- 消息可进行复制，且需要由客户端对其进行重复数据删除。

总而言之，这意味着您需要确保自己的异步作业以幂等方式编码并具备延迟弹性 [131]。头像的尺寸
整与替换正是幂等的典型实例，因为其经过两次处理仍将得到同样的结果。

最后，如果您的作业工作负载需要随时间推移进行规模伸缩（例如当在线用户更多时，头像上传
也将随之增加），请考虑使用 Auto Scaling 以启动竞价实例 [132]。Amazon SQS 提供多项指标供用
作为自动规模伸缩的触发条件，其中最常用的当数 ApproximateNumberOfMessagesVisible。可
消息数量基本上代表着您的队列积压量。举例来说，根据您每分钟所能处理的作业数量，您可以
其接近 100 时进行向上扩展，并在其低于 10 时进行向下扩展。欲了解更多与 Amazon SQS 及 Amaz
SNS 指标相关的信息，请参阅 Amazon CloudWatch 用户指南中的 Amazon SNS 指标与维度以及
Amazon SQS 指标与维度。[133]

FIFO 队列

虽然 Amazon SQS 的推荐使用方法强调应用程序本身应在工程与架构层面具备对重复及乱序状况
承受能力，但也有部分特定任务要求消息以明确的顺序进行接收与处理，这意味着消息重复绝对
可接受。举例来说，用户可能希望通过微交易对某一特定物品进行仅一次购买，这项操作就必须
到严格监控。

为了满足上述需求，先入先出（简称 FIFO）队列已经在特定 AWS 服务区内正式上线。FIFO 队
允许用户以有序且仅一次的方式处理消息内容。但正是由于对消息顺序与交付方式的严格要求，
FIFO 队列在实际使用中还存在其它一些限制。欲了解更多与 FIFO 队列相关的细节信息，请参阅
Amazon SQS 开发者指南中的 FIFO（先入先出）队列部分。[134]

其它队列选项

除了 Amazon SQS 与 Amazon SNS 之外，大家也可以利用其它多种方案在 Amazon EC2 实例上建
消息队列，具体包括 RabbitMQ、ActiveMQ 以及 Redis 等等。在使用这些方案时，您需要负责为
启动 EC2 实例组并加以配置，这部分内容不在本份白皮书的讨论范围之内。需要注意的是，运行
靠队列类似于运行高可用性数据库：您需要考虑高吞吐量磁盘（例如 Amazon EBS PIOPS）、快
冗余、复制以及故障转移等一切影响因素。保障定制化队列解决方案的正常运行与长期稳定需要
入大量时间，而且很可能在发生负载峰值时出现故障。

云成本

有了 AWS，您不再需要专门投入宝贵资源以构建价格昂贵的基础设施——包括采购服务器与软件可，或者租赁设备等等。在 AWS 的帮助下，您可以通过更低成本取代原本极为惊人的前期投入，时按照资源实际使用量计费。全部 AWS 服务皆以按需方式交付，且无需客户签订长期合约或者复的许可依赖关系。AWS 的核心优势包括：

- **按需实例** – AWS 针对超过 70 项云服务提供按使用量计费方案，这意味着游戏开发者能够速部署应用代码并以低廉的成本吸引游戏玩家。与供水及供电等公共服务类似，您只需要为自己实际使用量付费——一旦停止使用，其将不再产生任何额外费用。

- **保留实例** – Amazon EC2 等一部分 AWS 服务允许您以一到三年的协议期申购服务资源，时享受较按需实例更为实惠的成本折扣。以 Amazon EC2 为例，您可以选择无前期投入以享受更每小时成本，或者预先支付全部款项以包年方式使用（不再计算每小时费用）。

- **竞价实例** – Amazon EC2 竞价实例允许您对备用 Amazon EC2 容量投标，从而显著降低计资源使用成本。竞价实例适合那些规模可观且能够承受中断的工作负载; 批量处理与分析管道等对的游戏功能并不十分关键的用例最适合搭配竞价实例方案。

- **无服务器模式** – AWS Lambda 等其它服务拥有更为细化的计费方式 [135]。相较于按小时计费这类服务会以毫秒或者单次请求等极小单位作为计费标准。通过这种方式，您能够真正仅为自己用的资源付费，而不必承担由闲置服务带来的累积成本。

总结与未来展望

在本份白皮书中，我们探讨了大量基础性议题。下面，我们将重温其中的核心要点以及您在 AW
之上开启游戏运营旅程的简单步骤：

- 从小处着手，首先在 ELB 负载均衡器之后启动 2 个 EC2 实例。选择 Amazon RDS 或者 DynamoDB 作为数据库。考虑利用 Elastic Beanstalk 管理这套后台堆栈。
- 将游戏数据、资产以及补丁等二进制内容存储在 Amazon S3 之上。利用 Amazon S3 帮助的游戏服务器分担网络密集型工作负载。如果您需要立足全球分发此类资产，请考虑使用 CloudFront。
- 始终将您的 EC2 实例与数据库部署在多个可用区内以实现最佳可用性。作为起步，您可以实例分散在 2 个可用区当中。
- 随着服务器负载的增加，通过 ElastiCache 加入缓存机制。为您应用服务器所在的每个可内至少创建 1 个 ElastiCache 节点。
- 随着负载的持续增长，利用 Amazon SQS 或者 RabiitMQ 等其它队列方案将时间密集型操交由后台作业负责处理。如此一来，您的 EC2 应用实例及数据库将能够应对数量更为庞大的同时线玩家。
- 如果数据库性能出现问题，添加读取副本以分摊这部分读取/写入负载。评估是否能够添加 DyanmoDB 或者 Redis 等 NoSQL 存储方案以处理特定数据库任务。
- 在极端负载强度下，可以采取事件驱动型服务器或者分区数据库等更为先进的应对策略。而，除非确实必要，否则不要急于采用这些方法——因为其会显著提升开发、部署以及调试工作复杂性水平。

最后，Amazon Web Services 拥有一支致力于为游戏客户提供服务的商务与技术团队。若您需要系我们，请填写面向游戏开发者的 AWS 网站中的表单。[136]

贡献者

以下个人及组织为本份文件的编撰作出了贡献：

- Dhruv Thukral, Amazon Web Services 高级解决方案架构师
- Greg McConnel, Amazon Web Services 高级解决方案架构师
- Brent Nash, Amazon Game Studios 高级软件开发工程师
- Jack Hemion, Amazon Web Services 副解决方案架构师
- Keith Lafaso, Amazon Web Services 高级解决方案架构师

文档修订

日期	说明
2017 年 9 月	初版发布

备注

[1] https://aws.amazon.com/gaming/

[2] https://github.com/AFNetworking/AFNetworking

[3] http://api.jquery.com/jQuery.ajax/

[4] https://curl.haxx.se/

[5] http://en.cppreference.com/w/cpp/thread/async

[6] http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-listener-config.html

[7] http://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html

[8] https://aws.amazon.com/elasticbeanstalk/

[9] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/concepts.concepts.architecture.html

[10] https://aws.amazon.com/console/

[11] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/GettingStarted.html

[12] https://aws.amazon.com/free/

[13] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/applications.html

[14] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.environments.html

[15] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.CNAMESwap.html

[16] https://aws.amazon.com/about-aws/global-infrastructure/

[17] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.as.html

[18] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/configuring-https.html

[19] https://aws.amazon.com/s3/pricing/

[20] http://docs.aws.amazon.com/AmazonS3/latest/gsg/CreatingABucket.html

[21] http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html

[22] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/environment-resources.html

[23] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customize-environment-resources-elasticache.html

[24] https://aws.amazon.com/sqs/

[25] https://aws.amazon.com/sns/

[26] https://aws.amazon.com/blogs/aws/push-notifications-to-mobile-devices-using-amazon-sns/

[27] https://en.wikipedia.org/wiki/Representational_state_transfer

[28] http://rubyonrails.org/

[29] http://www.sinatrarb.com/

[30] https://github.com/ruby-grape/grape

[31] http://flask.pocoo.org/

[32] http://bottlepy.org/docs/dev/

[33] http://expressjs.com/

[34] https://github.com/restify/node-restify

[35] https://www.slimframework.com/

[36] https://silex.symfony.com/

[37] http://spring.io/

[38] https://jersey.github.io/

[39] https://github.com/gin-gonic/gin/

[40] https://aws.amazon.com/elasticbeanstalk/faqs/

[41] https://aws.amazon.com/elasticloadbalancing/

[42] http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-create-https-ssl-load-balancer.html

[43] http://docs.aws.amazon.com/acm/latest/userguide/acm-overview.html

[44] http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/using-domain-names-with-elb.html

45 https://aws.amazon.com/premiumsupport/

46 https://aws.amazon.com/articles/1636185810492479

47 http://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html

48 http://www.haproxy.org/

49 https://aws.amazon.com/marketplace/pp/B00B9KW32I/ref=portal_asin_url

50 http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/health-checks-creating-deleting.html

51 https://www.igvita.com/2012/10/31/simple-spdy-and-npn-negotiation-with-haproxy/

52 https://aws.amazon.com/opsworks/

53 http://docs.aws.amazon.com/autoscaling/latest/userguide/WhatIsAutoScaling.html

54 http://docs.aws.amazon.com/autoscaling/latest/userguide/WhatIsAutoScaling.html

55 http://httpd.apache.org/docs/2.2/programs/ab.html

56 https://github.com/httperf/httperf

57 http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scale-based-on-demand.html

58 http://docs.aws.amazon.com/opsworks/latest/userguide/workinginstances-autoscaling.html

59 http://docs.aws.amazon.com/opsworks/latest/userguide/workinginstances-autoscaling.html

60 https://help.ubuntu.com/community/CloudInit

61 https://aws.amazon.com/amazon-linux-ami/

62 https://aws.amazon.com/architecture/

63 https://github.com/eventmachine/eventmachine

64 http://www.gevent.org/

[65] http://twistedmatrix.com/trac/

[66] https://nodejs.org/en/

[67] https://socket.io/

[68] https://github.com/caolan/async

[69] http://www.erlang.org/

[70] https://netty.io/wiki/index.html

[71] https://do.awsstatic.com/whitepapers/optimizing-multiplayer-game-server-performance-on-aws.pdf

[72] https://aws.amazon.com/blogs/gamedev/fitting-the-pattern-serverless-custom-matchmaking-with-amazon-gamelift/

[73] http://docs.aws.amazon.com/sns/latest/dg/welcome.html

[74] http://docs.aws.amazon.com/sns/latest/dg/CreateTopic.html

[75] http://docs.aws.amazon.com/sns/latest/dg/SNSMobilePush.html

[76] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_GettingStarted.html#CHAP_GettingStarted.CreatingConnecting.MySQL

[77] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html#USER_PIOPS

[78] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithParamGroups.html

[79] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Appendix.MySQL.CommonDBATasks.html

[80] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ModifyInstance.MySQL.html

[81] http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.html

[82] https://redis.io/topics/data-types

[83] https://redis.io/topics/data-types-intro

84 https://redis.io/topics/benchmarks

85 https://aws.amazon.com/elasticache/redis/

86 https://do.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf

87 https://docs.mongodb.com/manual/reference/command/findAndModify/

88 https://www.mongodb.com/

89 https://do.awsstatic.com/whitepapers/AWS_NoSQL_MongoDB.pdf

90 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ProvisionedThroughput.html

91 http://docs.aws.amazon.com/AWSRubySDK/latest/index.html

92 http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/index.html

93 http://boto.readthedocs.io/en/latest/dynamodb_tut.html

94 http://docs.aws.amazon.com/amazondynamodb/latest/gettingstartedguide/Welcome.html

95 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AutoScaling.html

96 http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

97 https://aws.amazon.com/blogs/aws/optimizing-provisioned-throughput-in-amazon-dynamodb/

98 http://memcached.org/

99 https://redis.io/

100 https://aws.amazon.com/elasticache/

101 http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/WhatIs.html

102 http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/CacheMetrics.html

103 http://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html

104 http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/ParameterGroups.html

105 http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/CacheMetrics.WhichShouldIMonitor.html

106 https://aws.amazon.com/s3/

107 http://docs.aws.amazon.com/AmazonS3/latest/gsg/CreatingABucket.html

108 https://aws.amazon.com/cloudfront/

109 https://aws.amazon.com/cloudfront/

110 https://aws.amazon.com/cloudfront/details/#faq

111 https://aws.amazon.com/contact-us/aws-sales/

112 https://en.wikipedia.org/wiki/HTTP_ETag

113 https://en.wikipedia.org/wiki/HTTP_ETag#Typical_usage

114 http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/RequestAndResponseBehaviorS3Origin.html

115 http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingHTTPPOST.html

116 http://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectPUT.html

117 http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingHTTPPOST.html

118 http://docs.aws.amazon.com/AmazonS3/latest/dev/NotificationHowTo.html

119 https://aws.amazon.com/emr/

120 https://aws.amazon.com/redshift/

121 https://aws.amazon.com/big-data/

122 https://d0.awsstatic.com/whitepapers/aws-amazon-emr-best-practices.pdf

123 https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-tips-for-amazon-athena/

[124] http://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html

[125] https://aws.amazon.com/premiumsupport/

[126] http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features-managing-env-tiers.html

[127] https://aws.amazon.com/sqs/

[128] http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-getting-started.html

[129] http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/throughput.html

[130] https://aws.amazon.com/ec2/spot/

[131] https://en.wikipedia.org/wiki/Idempotence

[132] http://docs.aws.amazon.com/autoscaling/latest/userguide/US-SpotInstances.html

[133] http://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/sns-metricscollected.html

[134] http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/FIFO-queues.html

[135] https://aws.amazon.com/lambda/

[136] https://aws.amazon.com/gaming/