无服务器应用程序剖析

AWS 架构完善的框架

2019 年 12 月

声明

客户有责任对本文档中的信息进行独立评估。本文档：(a) 仅供参考，(b) 代表 AWS 当前的产品服务和实践，如有变更，恕不另行通知，以及 (c) 不构成 AWS 及其附属公司、供应商或授权商的任何承诺或保证。AWS 产品或服务均“按原样”提供，没有任何明示或暗示的担保、声明或条件。AWS 对其客户的责任和义务由 AWS 协议决定，本文档与 AWS 和客户之间签订的任何协议无关，亦不影响任何此类协议。

目录

摘要

本文档描述了适用于 AWS 架构完善的框架的**无服务器应用程序剖析**。文档涵盖了常见的无服务器应用程序场景，并指明了确保工作负载架构设计符合最佳实践的关键元素。

简介

AWS 架构完善的框架能够帮助您认识到您在 AWS 上构建系统时所做决策的优缺点。[1] 通过使用此框架，您将了解在云中设计和运行可靠、安全、高效且经济实惠的系统的架构最佳实践。它提供了一种方法，使您能够根据最佳实践持续衡量架构，并确定需要改进的方面。我们相信，拥有架构完善的系统能够大大提高实现业务成功的可能性。

在这篇“剖析”中，我们重点介绍如何在 AWS 云中对**无服务器应用程序工作负载**进行设计、部署和架构设计。为简洁起见，我们仅提供架构完善的框架中特定于无服务器工作负载的详细信息。在设计架构时，您还应考虑本文档中未包括的最佳实践和问题。我们建议您阅读 AWS 架构完善的框架白皮书。[2]

本文档面向技术人员，例如首席技术官 (CTO)、架构师、开发人员和运维团队成员。阅读本文档后，您将了解在设计适用于无服务器应用程序的架构时可以采用的 AWS 最佳实践和策略。

定义

AWS 架构完善的框架建立在五大支柱的基础上，它们是卓越运营、安全性、可靠性、性能效率和成本优化。对于无服务器工作负载，AWS 提供了多个核心组件（无服务器和非无服务器），使您可以为无服务器应用程序设计可靠的架构。在此部分中，我们将概述将在本文档中使用的服务。构建无服务器工作负载时，应考虑七个方面：

- 计算层
- 数据层
- 消息收发和流式处理层
- 用户管理和身份层
- 边缘层
- 系统监控和部署
- 部署方法

计算层

工作负载的计算层管理来自外部系统的请求、控制访问权限并确保对请求进行适当的授权。其中包含将用于部署和执行您的业务逻辑的运行时环境。

AWS Lambda 允许您在托管平台上运行无状态无服务器应用程序，该平台支持微服务架构、部署和功能层的执行管理。

借助 **Amazon API Gateway**，您可以运行一个完全托管式 REST API，它可与 Lambda 集成以执行您的业务逻辑，并且包含流量管理、授权和访问权限控制、监控以及 API 版本控制功能。

AWS Step Functions 可编排无服务器工作流，包括协调、状态和函数链，还能整合 Lambda 执行限制内并不支持的长时间运行的执行，它将此类执行分解为多个步骤或调用在 Amazon Elastic Compute Cloud (Amazon EC2) 实例或本地运行的工作程序。

数据层

工作负载的数据层管理系统中的持久性存储。它提供了一种安全的机制来存储您的业务逻辑需要的状态。它提供了一种机制来触发事件以响应数据更改。

Amazon DynamoDB 可提供一个托管的 NoSQL 数据库来支持持久性存储，帮助您构建无服务器应用程序。结合 **DynamoDB Streams**，您可以通过调用 Lambda 函数，以近乎实时的方式响应 DynamoDB 表中的更改。**DynamoDB Accelerator** (DAX) 为 DynamoDB 添加了高度可用的内存中缓存，将性能提升了多达 10 倍，从毫秒级提升到微秒级。

使用 **Amazon Simple Storage Service** (Amazon S3)，您可以通过提供高度可用的键值存储（可通过 **Amazon CloudFront** 之类的 Content Delivery Network (CDN) 从中提供静态资产）来构建无服务器的 Web 应用程序和网站。

Amazon Elasticsearch Service (Amazon ES) 可以轻松部署、保护、操作和扩展 Elasticsearch，以实现日志分析、全文搜索、应用监控等功能。Amazon ES 是一种完全托管式服务，可提供一个搜索引擎和几种分析工具。

AWS AppSync 是一种托管式 GraphQL 服务，具有实时和脱机功能以及可简化应用程序开发的企业级安全控件。AWS AppSync 提供了数据驱动的 API 和一致的编程语言，让应用程序和设备可以连接到 DynamoDB、Amazon ES 和 Amazon S3 等服务。

消息收发和流式处理层

工作负载的消息收发层管理组件之间的通信。流式处理层管理流式数据的实时分析和处理。

Amazon Simple Notification Service (Amazon SNS) 使用针对微服务、分布式系统和无服务器应用程序的异步事件通知和移动推送通知，为发布/订阅模式提供完全托管的消息收发服务。

Amazon Kinesis 可轻松收集、处理和分析实时流式数据。借助 **Amazon Kinesis Data Analytics**，您可以运行标准 SQL 或使用 SQL 构建完整的流式传输应用。

Amazon Kinesis Data Firehose 捕获、转换流式数据并将其加载到 Kinesis Data Analytics、Amazon S3、Amazon Redshift 和 Amazon ES 中，从而可以使用现有的商业智能工具进行近乎实时的分析。

用户管理和身份层

工作负载的用户管理和身份层为工作负载接口的内部和外部客户提供身份、身份验证和授权。

借助 **Amazon Cognito**，您可以轻松地向无服务器应用程序添加用户注册、登录和数据同步功能。**Amazon Cognito** 用户池提供内置的登录屏幕以及与 Facebook、Google、Amazon 和安全性声明标记语言 (SAML) 的联合。**Amazon Cognito Federated Identities** 让您安全地提供对无服务器架构中 AWS 资源的限定范围的访问。

边缘层

工作负载的边缘层管理表示层以及与外部客户的连接。它为身处不同地理位置的外部客户提供了一种有效的交付方式。

Amazon CloudFront 提供了一个 CDN，它能以低延迟和高传输速度安全地交付 Web 应用程序内容和数据。

系统监控和部署

工作负载的系统监控层通过相关指标管理系统可见性，并打造对其在一定时间内的运行和行为方式的情境感知。部署层定义了如何通过发布管理流程促进工作负载更改。

借助 **Amazon CloudWatch**，您可以访问所使用的所有 AWS 服务上的系统指标、整合系统和应用级别日志，并创建业务关键绩效指标 (KPI) 来作为满足您特定需求的自定义指标。它提供控制面板，以及可以触发平台上自动操作的警报。

AWS X-Ray 通过提供分布式跟踪和服务映射来分析和调试无服务器应用程序，从而通过可视化端到端请求轻松识别性能瓶颈。

AWS 无服务器应用程序模型 (AWS SAM) 是 AWS CloudFormation 的扩展，用于打包、测试和部署无服务器应用程序。在本地开发 Lambda 函数时，AWS SAM CLI 还可以实现更快的调试周期。

部署方法

微服务架构中部署的最佳实践是确保更改不会破坏使用者的服务合约。如果 API 拥有者进行更改而违反了服务合同，而使用者没有为此做好准备，则可能会发生问题。

知道哪些使用者正在使用您的 API 是确保部署安全的第一步。收集有关使用者及其使用情况的元数据，使您可以做出有关变更影响的数据驱动型决策。API 密钥是一种捕获有关 API 使用者/客户的元数据的有效方法，如果对 API 进行了重大更改，API 密钥通常用作一种联系方式。

某些希望针对重大更改采用风险规避方法的客户可以选择克隆 API 并将客户路由到其他子域（例如 v2.my-service.com），以确保不影响现有的客户。尽管此方法可以通过新的服务合约进行新的部署，但要权衡的是维护双 API（以及后续后端基础设施）的额外开销需求。

该表显示了不同的部署方法：

部署	使用者影响	回滚	事件模型系数	部署速度
一次性	一次性	重新部署旧版本	以较低的并发速率部署任何事件模型	即时

蓝/绿	一次性，预先进行一定水平的生产环境测试	将流量恢复到以前的环境	适用于中等并发工作负载下的异步和同步事件模型	几分钟到几小时的验证，然后立即提供给客户
Canary 部署/线性部署	典型的初始流量转移为 1–10%，然后分阶段循序渐进或一次性完成	将 100% 的流量恢复到以前的部署	更适合高并发工作负载	几分钟到几小时

一次性部署

一次性部署需要在现有配置的基础上做出更改。这种部署方式的一个优势是，对数据存储（例如关系数据库）的后端更改在更改周期内协调事务所需的精力要少得多。尽管这种类型的部署方式较为省力，并且对低并发模型的影响很小，但它在回滚时会增加风险，通常会导致停机。使用此部署模型的一个示例场景是在用户影响最小的开发环境中。

蓝绿部署

另一种流量转移模式是启用蓝/绿部署。这种接近零停机的发布方式使流量可以转移到新的实时环境（绿色），同时在需要回滚的情况下仍保持旧的生产环境（蓝色）处于"温"状态。由于 API 网关允许您定义将多少百分比的流量转移到特定环境；这种部署方式可能是一种有效的技术。由于蓝/绿部署旨在减少停机时间，因此许多客户采用这种模式处理生产变更。

遵循无状态和幂等最佳实践的无服务器架构适合这种部署方式，因为它与底层基础设施没有紧密关联。您应该使这些部署偏向于较小的增量更改，以便在必要时可以轻松地回滚到工作环境。

您需要适当的指标来了解是否需要回滚。作为最佳实践，我们建议客户使用 CloudWatch 高解析度指标，该指标可以每隔 1 秒执行一次监控，并快速捕获下降趋势。结合使用 CloudWatch 警报，即可启用快速回滚。可以在 API 网关、Step Functions、Lambda（包括自定义指标）和 DynamoDB 上捕获 CloudWatch 指标。

Canary 部署

Canary 部署是您在受控环境中利用软件的新版本并实现快速部署周期的一种循序渐进的方式。Canary 部署涉及到部署对新更改的少量请求，以分析对少量用户的影响。由于您不再需要担心新部署的基础设施的预置和扩展，因此 AWS 云有助于促进这种采用。

使用 API 网关中的 Canary 部署，您可以将更改部署到后端终端节点（例如 Lambda），同时仍为使用者维护相同的 API 网关 HTTP 终端节点。此外，您还可以控制将多少百分比的流量路由到新的部署并实现受控制的流量切换。Canary 部署的实际应用场景可能包括新网站。在将所有流量转移到新部署之前，您可以监控少量最终用户的点击访问率。

Lambda 版本控制

与所有软件一样，通过保持版本控制，您可以快速查看以前能正常运行的代码，并且能够在新部署失败的情况下恢复到以前的版本。Lambda 允许您为单个 Lambda 函数发布一个或多个不可变版本；这样可确保以前的版本不会发生更改。每个 Lambda 函数版本都有一个唯一的 Amazon 资源名称 (ARN)，并且新版本的更改可以记录在 CloudTrail 中，因此可以审计。作为生产中的最佳实践，客户应启用版本控制以最充分地利用可靠的架构。

为了简化部署操作并降低错误风险，Lambda 别名启用了 Lambda 函数在开发工作流程中的不同变体，例如开发、测试版和生产。例如，当 API 网关与 Lambda 的集成指向生产别名的 ARN 时。生产别名将指向 Lambda 版本。这项技术的价值在于，在将新版本提升到实际运行环境时，它可以实现安全部署，因为调用方配置中的 Lambda 别名保持静态，因此只需进行较少的更改。

一般设计原则

架构完善的框架定义了一系列一般性设计原则，以促进良好的无服务器应用程序云端设计：

- **快速、简单、单一**：功能简洁、简短、单一用途，其环境可符合其请求的生命周期。事务能更有效地掌控成本，因此更快执行速度的方法是首选方法。
- **考虑并发请求数，而不是总请求数**：无服务器应用程序利用并发模型，并根据并发性评估设计级别的权衡折衷。

- **不共享任何内容**：函数运行时环境和底层基础设施均为短期存在，因此不能保证诸如临时存储之类的本地资源。可以在状态机执行生命周期内操控状态，持久性存储对于高持久性需求而言属于首选方法。
- **假定无硬件关联**：底层基础设施可能会发生变化。例如，利用 CPU 标记为并非一致可用的与硬件无关的代码或依赖项。
- **使用状态机而不是函数来协调您的应用**：在代码内链接 Lambda 执行以协调您的应用工作流程会带来整体化、紧密耦合的应用。不要采用这种方式，而是应该使用状态机来协调事务和通信流。
- **使用事件触发事务**：诸如写入新的 Amazon S3 对象或对数据库进行更新之类的事件允许执行事务以响应业务功能性需求。这种异步事件行为通常与使用者无关，会驱动即时处理以确保精益的服务设计。
- **针对故障和重复项的设计**：请求/事件触发的操作必须是幂等的，因为故障可能不时发生，给定的请求/事件可以多次传递。包含适当的下游调用重试。

场景

在本节中，我们介绍了许多无服务器应用程序中常见的五个关键场景，以及它们如何影响 AWS 上无服务器应用程序工作负载的设计和架构。我们将介绍针对每种场景做出的假设、设计的通用驱动因素，以及应如何实现这些场景的参考架构。

RESTful 微服务

构建微服务时，您要考虑如何将业务环境作为可重用服务交付给您的使用者。具体的实现将针对单个用例进行量身定制，但是微服务有几个共同的主题，以确保您的实现是安全、有弹性的，并且能够为您的客户提供最佳体验。

在 AWS 上构建无服务器微服务使您不仅可以充分利用无服务器功能本身，还可以使用其他 AWS 服务和功能，以及 AWS 和 AWS 合作伙伴网络 (APN) 工具的生态系统。无服务器技术建立在容错基础设施之上，使您能够为关键任务工作负载构建可靠的服务。工具生态系统使您可以简化构

建、自动化任务、协调依赖关系以及监控和管理微服务。最后，AWS 无服务器工具采用按使用量付费的模式，使您能够根据业务情况发展服务，并在进入阶段和非高峰时间降低成本。

特征：

- 您需要一个安全、易于操作的框架，它易于复制，具有高水平的弹性和可用性。
- 您想要记录利用率和访问模式，以持续改进后端以支持客户使用。
- 您想要尽可能多地利用您的平台托管服务，这减少了与管理常见平台（包括安全性和可扩展性）相关的繁重任务。

参考架构

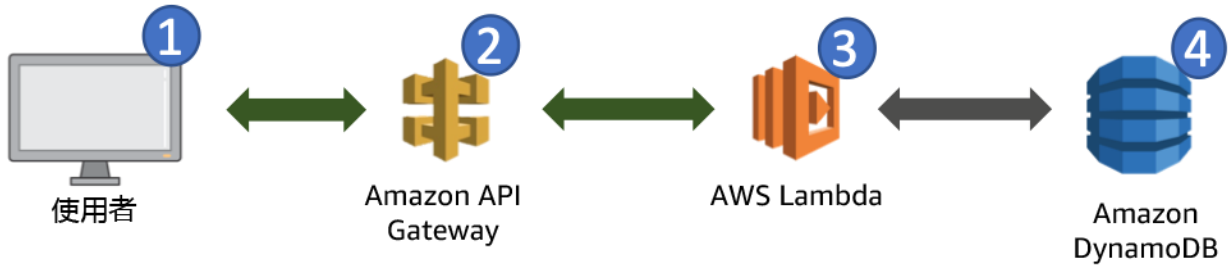

图 1：RESTful 微服务的参考架构

1. **客户**通过进行 HTTP API 调用来利用您的微服务。在理想情况下，您的消费者应与您的 API 签订紧密的服务合同，以实现对服务级别和更改控制的一致期望。
2. **Amazon API Gateway** 主机对客户的 RESTful HTTP 请求和响应。在这种情况下，API 网关提供内置授权、限制、安全性、容错能力、请求/响应映射和性能优化。
3. **AWS Lambda** 包含处理传入 API 调用并将 DynamoDB 用作持久性存储的业务逻辑。
4. **Amazon DynamoDB** 持续存储微服务数据，并根据需求扩展。由于微服务的设计通常能很好地执行一项操作，定期合并无模式 NoSQL 数据存储。

配置说明：

- 利用 API 网关日志记录来了解微服务消费者访问行为的可见性。此信息在 Amazon CloudWatch Logs 中可见，并且可通过 Log Pivots 快速查看，在 CloudWatch Logs Insights 中分析或输入到其他可搜索引擎，如 Amazon ES 或 Amazon S3（使用 Amazon Athena）。所提供的信息提供了关键可见性，例如：

- 了解常见客户位置，这可能会因后端的距离而发生地理变化
- 了解客户输入请求对数据库分区的影响
- 了解异常行为的语义，这可以是安全标志
- 了解错误、延迟和高速缓存命中/错过以优化配置

此模型提供了易于部署和维护的框架，并提供了一个安全的环境，可随着您的需求的增长而扩展。

Alexa 技能

Alexa 技能套件使开发人员能够通过构建自然且高效的语音和视频体验来扩展 Alexa 的功能。成功的技能为习惯使然，用户成了常客，因为该套件提供了独特的性能，以全新的新颖且无摩擦的方式提供价值。

用户感到沮丧的最大原因是，该技能无法按预期方式操作，可能需要进行多次交互才能完成所需的任务。首先，必须设计语音交互模型，然后从该模型向后工作，因为某些用户可能会说得太少或太多，或者可能是您不希望听到的内容。语音设计过程包括创建、编写脚本和规划预期和非预期的语句。

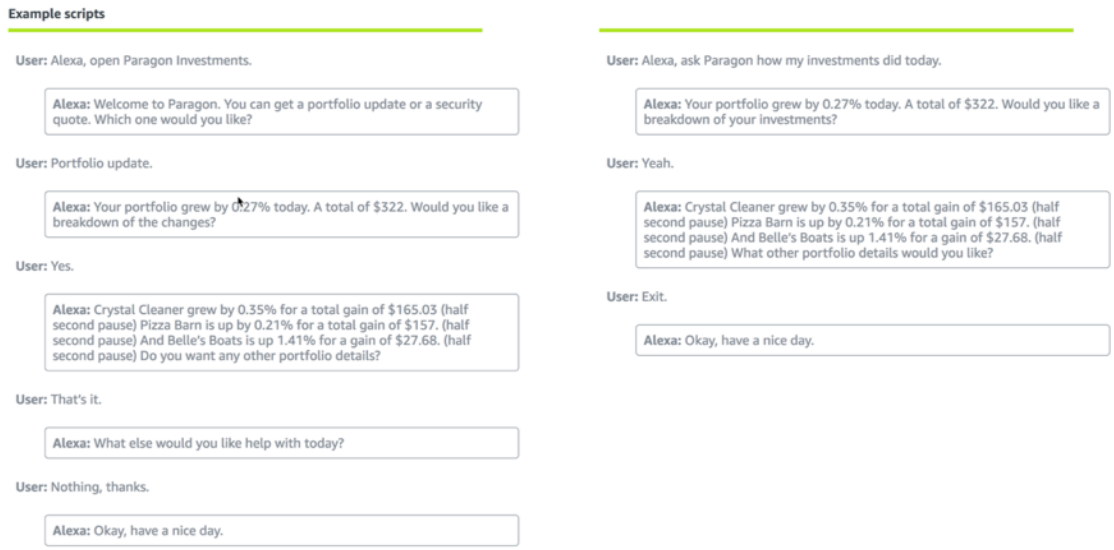

图 2：Alexa 技能示例设计脚本

考虑到基本脚本，您可以在开始构建技能之前使用以下技巧：

- **概述完成操作的最短路线**
    - 完成操作的最短路线通常是用户一次性提供所有信息和插槽、账户已链接（如果相关）和其他前提条件在一次调用该技能时得到满足。
- **概述备选路径和决策树**
    - 通常，用户所说的内容不包括完成请求所需的所有信息。在流程中，确定备选路径和用户决策。
- **概述系统逻辑必须做出的幕后决策**
    - 确定幕后系统决策，例如新用户或返回用户。后台系统检查可能会更改用户遵循的流程。
- **概述技能将如何帮助用户**
    - “帮助”中包含明确的说明，说明用户可以使用该技能执行哪些操作。根据技能的复杂性，“帮助”可能会提供一个简单的响应或多个响应。
- **概述账户链接流程（如有）**
    - 确定账户链接所需的信息。您还需要确定在账户链接尚未完成时技能将如何响应。

特征：

- 您想要创建一个完整的无服务器架构，而无需管理任何实例或服务器。
- 您想要内容尽可能与您的技能分离。
- 您想要以 API 的形式提供高效的语音体验，以优化各种 Alexa 设备、地区和语言的开发。
- 您想要获得弹性，能够上下扩展，以满足用户的需求并处理意外的使用模式。

参考架构

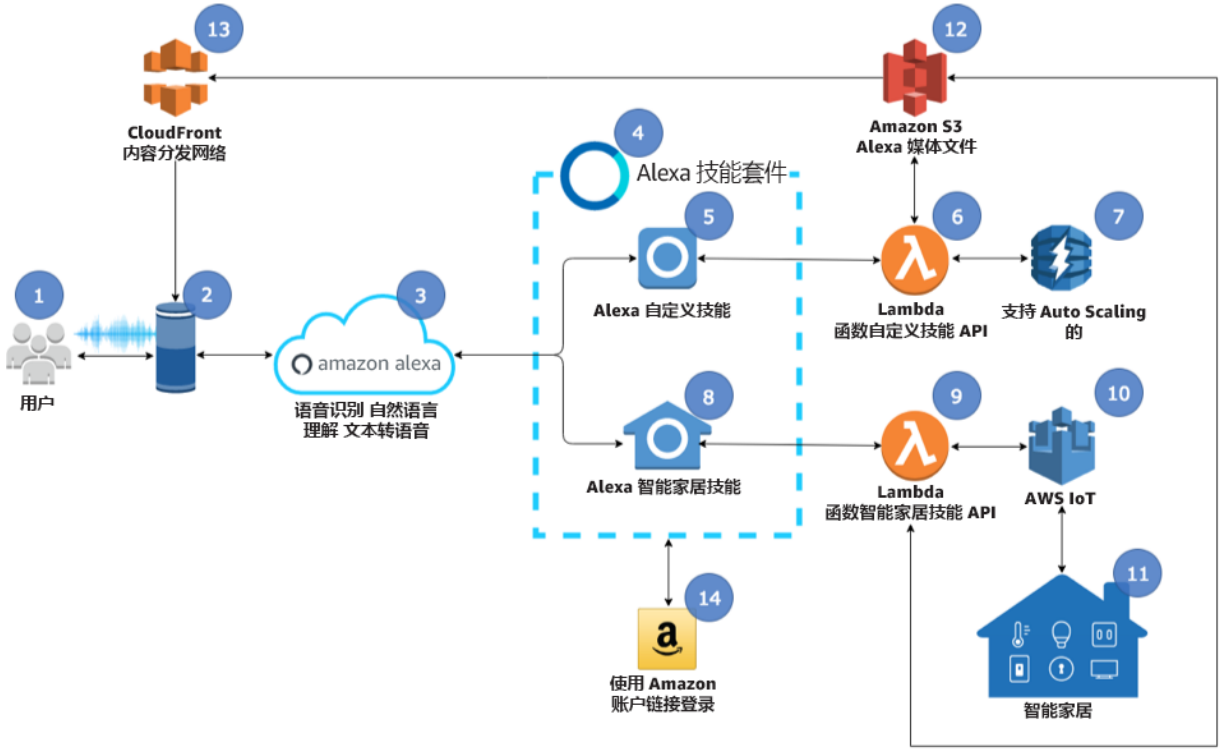

图 3：Alexa 技能参考架构

1. **Alexa 用户**通过使用语音作为主要交互方法与支持 Alexa 的设备通话与 Alexa 交互。
2. **支持 Alexa 的设备**会监听唤醒词，并在识别到唤醒词后立即激活。支持的唤醒词是 Alexa、计算机和 Echo。
3. **Alexa 服务**代表您的 Alexa 技能执行通用语音语言理解 (SLU) 处理，包括自动语音识别 (ASR)、自然语言理解 (NLU) 和文本至语音 (TTS) 转换。
4. **Alexa 技能套件 (ASK)** 是自助服务 API、工具、文档和代码示例的集合，使您可以快速轻松地向 Alexa 添加技能。ASK 是值得信赖的 AWS Lambda 触发器，可实现无缝集成。
5. **Alexa 自定义技能**让您能够控制用户体验，让您能够构建自定义交互模型。它是最灵活的技能类型，但也是最复杂的。
6. 使用 Alexa 技能套件的 **Lambda 函数**，使您能够无缝地培养技能，避免不必要的复杂性。使用它，您可以处理 Alexa 服务发送的不同类型的请求，并生成语音响应。

7. **DynamoDB 数据库**提供 NoSQL 数据存储，可随您的门槛的使用灵活扩展。技能通常使用该数据库来保持用户状态和会话。
8. **Alexa 智能家居技能**允许您使用智能家居 API 控制设备，如灯光、恒温器、智能电视等。智能家居技能可以更简单地构建自定义技能，因为"不会"让您控制互动模式。
9. **Lambda 函数**用于响应来自 Alexa 服务的设备发现和控制请求。开发人员使用它来控制多种设备，包括娱乐设备、摄像头、照明、恒温器、锁等。
10. **AWS 物联网 (IoT)** 允许开发人员安全地将设备连接到 AWS，并控制 Alexa 技能与设备之间的交互。
11. 支持 Alexa 的**智能家居**有数量不限的 IoT 连接设备，可以接收和响应来自 Alexa 技能的指令。
12. **Amazon S3** 存储您的技能静态资产，包括图片、内容和媒体。它的内容使用 CloudFront 安全提供。
13. **Amazon CloudFront 内容分发网络** (CDN) 提供 CDN，可更快地为地理分布式移动用户提供内容，并在 Amazon S3 中包含静态资产的安全机制。
14. 当您的技能必须通过另一个系统验证时，此时将需要**账户链接**。此操作将 Alexa 用户与其他系统中的特定用户相关联。

配置说明：

- 通过根据 JSON 模式验证技能发送至 Alexa 的所有可能 Alexa 智能家居消息，验证智能家居请求和响应有效负载。
- 确保您的 Lambda 函数超时时间少于 8 秒，并且可以在该时间范围内处理请求。（Alexa 服务超时为 8 秒。）
- 创建 DynamoDB 表时，遵循最佳实践 [7]。当您不确定所需的读/写容量时，请使用按需表。否则，选择已启用自动缩放功能的已配置容量。对于需要大量准备工作的技能，DynamoDB 加速器 (DAX) 可以大大缩短响应时间。
- 账户链接可提供存储在外部系统中的用户信息。使用该信息为您的用户提供背景和个性化体验。Alexa 有关于账户链接的指南，可提供无摩擦体验。

- 使用技能测试版测试工具收集有关技能发展的早期反馈，并用于技能版本控制，以减少对已上线技能的影响。
- 使用 ASK CLI 自动进行技能开发和部署。

移动后端

用户越来越希望他们的移动应用程序能提供快速、一致且功能丰富的用户体验。同时，移动用户模式是动态的，峰值使用率不可预测，并且通常具有全球影响。

移动用户不断增长的需求意味着应用程序需要一套丰富的移动服务，这些服务可以无缝协作，而不会牺牲后端基础设施的控制和灵活性。默认情况下，移动应用程序中预期的某些功能：

- 能够查询、更改和订阅数据库更改
- 连接时数据和带宽优化的离线持久性
- 在应用程序中搜索、筛选和发现数据
- 用户行为分析
- 通过多个渠道（推送通知、短信、电子邮件）发送定向消息
- 丰富的内容，如图像和视频
- 跨多个设备和多个用户的数据同步
- 用于查看和处理数据的细化授权控件

在 AWS 上构建无服务器移动后端，使您能够提供这些功能，同时自动管理可扩展性、弹性，并以高效且具有成本效益的方式提供可用性。

特征：

- 您要从客户端控制应用程序数据行为，并明确选择要从 API 中获取的数据
- 您想要您的业务逻辑尽可能与您的移动应用程序分离。
- 您想要以 API 的形式提供业务功能，以优化跨多个平台的开发。

- 您正在寻求利用托管服务来减少维护移动后端基础设施的无差别繁重工作量，同时提供高级别的可扩展性和可用性。
- 您想要根据实际用户需求和支付闲置资源来优化移动后端成本

参考架构

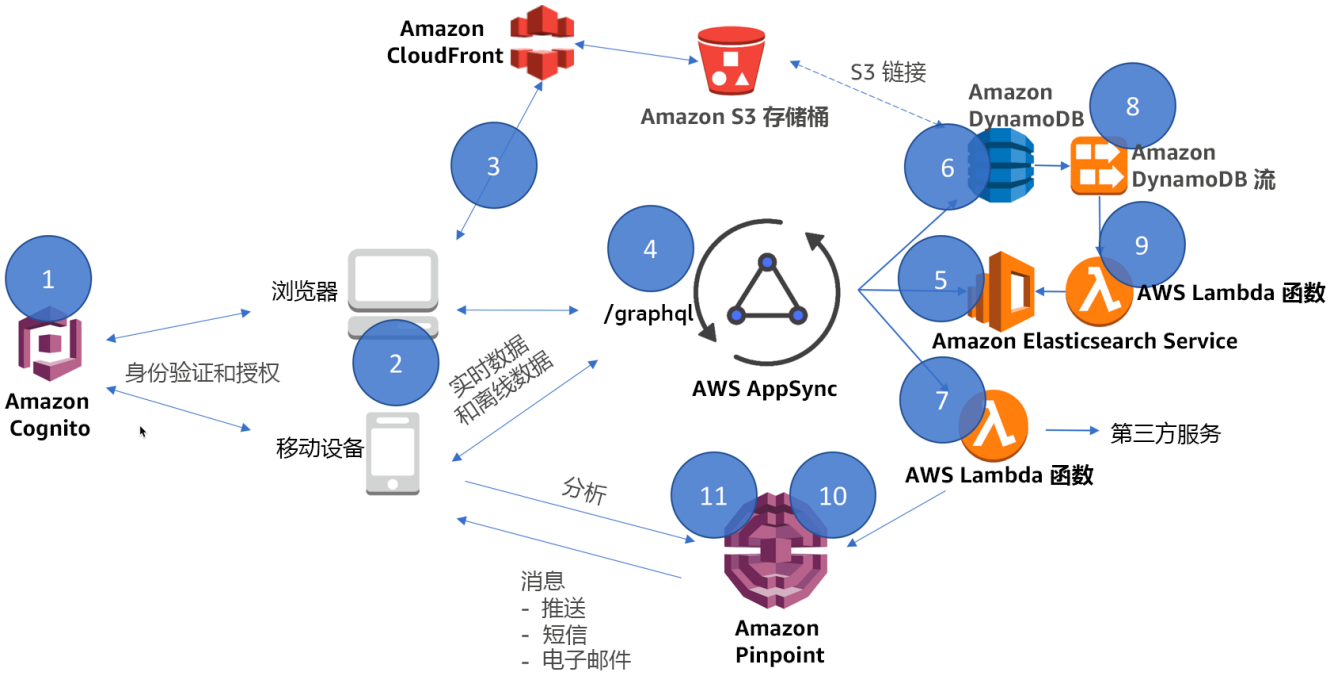

图 4：移动后端参考架构

1. **Amazon Cognito** 用于用户管理和您的移动应用程序的身份提供商。此外，它还允许移动用户利用现有的社交身份，如 Facebook、Twitter、Google+ 和 Amazon 进行登录。
2. **移动用户**通过针对 AWS AppSync 和 AWS 服务 API（例如 Amazon S3 和 Amazon Cognito）执行 GraphQL 操作与移动应用程序后端交互。
3. **Amazon S3** 存储移动应用程序静态资产，包括特定移动用户数据，如配置文件图像。其内容通过 CloudFront 安全提供。
4. **AWS AppSync** 托管面向移动用户的 GraphQL HTTP 请求和响应。在这种情况下，当设备已连接，来自 AWS AppSync 的数据是实时的，数据也可离线使用。此情形的数据源是 **Amazon DynamoDB**、**Amazon Elasticsearch Service** 或 **AWS Lambda** 函数

5. **Amazon Elasticsearch Service** 充当您的移动应用程序和分析的主要搜索引擎。
6. **DynamoDB** 为您的移动应用程序提供持久性存储，包括通过生存时间 (TTL) 功能使非活动移动用户不需要的数据过期的机制。
7. **Lambda** 函数处理与其他第三方服务的交互，或调用其他 AWS 服务进行自定义流，这可以是 GraphQL 对客户端响应的一部分。
8. **DynamoDB 流**捕获项目级更改，并启用 Lambda 函数来更新其他数据源。
9. **Lambda** 函数管理 DynamoDB 和 Amazon ES 之间的流数据，允许客户合并数据源逻辑 GraphQL 类型和操作。
10. **Amazon Pinpoint** 从客户处获取分析，包括用户会话和自定义指标以获取应用程序见解。
11. **Amazon Pinpoint** 根据收集到的分析，将消息发送给所有用户/设备或目标子集。可以使用推送通知、电子邮件或 SMS 信道自定义和发送消息。

配置说明：

- 性能测试[3]您的 Lambda 函数具有不同的内存和超时设置，以确保您使用最适合作业的资源。
- 在创建 DynamoDB 表时遵循最佳实践[4]，并考虑让 AWS AppSync 从 GraphQL 模式进行自动预置，此架构将使用分布良好的哈希键并为您的操作创建索引。确保计算读/写容量和表分区，以确保合理的响应时间。
- 使用 AWS AppSync 服务器端数据缓存优化您的应用程序体验，由于所有后续的 API 查询请求都将从缓存中返回，这意味着除非 TTL 到期，否则不会直接联系数据源。
- 在管理 Amazon ES 域时遵循最佳实践[5]。此外，Amazon ES 还提供了大量关于在此处同样适用的分区和访问模式的设计指南[6]。
- 使用在解析程序中配置的 AWS AppSync 的细粒度访问控制，根据需要将 GraphQL 请求过滤到每用户或组级别。这可以应用于 AWS Identity and Access Management (IAM) 或使用 AWS AppSync 的 Amazon Cognito 用户池授权。
- 使用 AWS Amplify 和 Amplify CLI 来编写应用程序并将其与多个 AWS 服务集成。Amplify 控制台还负责部署和管理堆栈。

对于需要近乎无业务逻辑的低延迟要求，Amazon Cognito 联合身份可以提供范围限定的凭证，以便您的移动应用程序可以直接与 AWS 服务通信，例如，在上载用户的配置文件图片时，从 Amazon S3 检索范围限定为用户的元数据文件等等

流式处理

接收和处理实时流数据需要可扩展性和低延迟，以支持各种应用程序，如活动跟踪、事务订单处理、点击流分析、数据清理、指标生成、日志筛选、索引、社交媒体分析和物联网设备数据遥测和计量。这些应用程序通常具有峰值，每秒处理数千个事件。

使用 AWS Lambda 和 Amazon Kinesis，您可以构建无需配置或管理服务器即可自动扩展的无服务器流进程。AWS Lambda 处理的数据可以存储在 DynamoDB 中，并在以后进行分析。

特征：

- 您想要创建一个完整的无服务器架构，而无需管理任何实例或服务器来处理流数据。
- 您想要使用 Amazon Kinesis 创建器库 (KPL) 从数据生产者的角度处理数据提取。

参考架构

此处我们将介绍一个常见流处理场景，这是用于分析社交媒体数据的参考架构。

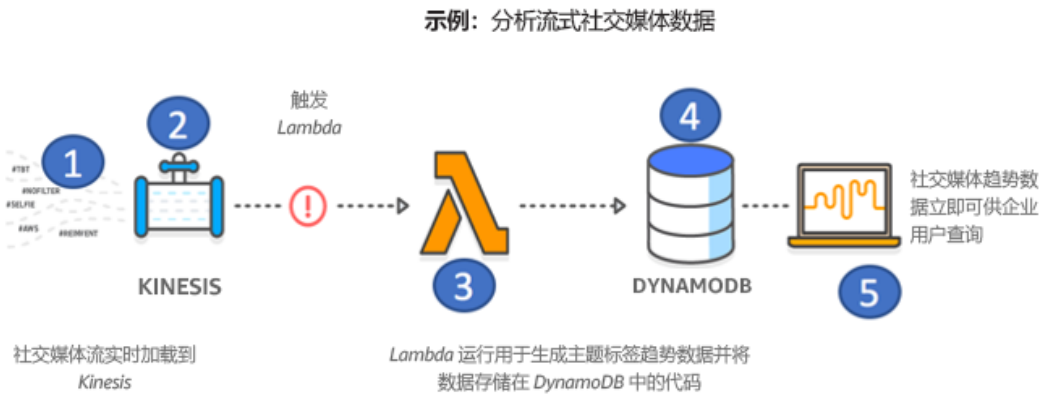

图 5：用于流处理的参考架构

1. **数据创建器**使用 Amazon Kinesis 创建器库 (KPL) 将社交媒体流数据发送到 Kinesis 流。也可以使用利用 Kinesis API 的 Amazon Kinesis 代理和自定义数据创建器。

2. **Amazon Kinesis 流**收集、处理并分析数据创建器生成的实时流数据。提取到数据流中的数据可以由使用者处理，在本例中是 Lambda
3. **AWS Lambda** 充当数据流的使用者，该流以单个事件/调用的形式接收提取的数据数组。Lambda 函数将执行进一步处理。转换后的数据将存储在持久性存储中，在本例中是 DynamoDB。
4. **Amazon DynamoDB** 提供快速灵活的 NoSQL 数据库服务，包括可与 AWS Lambda 集成的触发器，以便在其他地方提供此类数据。
5. **业务用户**利用 DynamoDB 上的报告界面从社交媒体趋势数据中收集见解

配置说明：

- 当对 Kinesis 流进行重新分区以适应更高的提取率时遵循最佳实践[7]。流处理的并发性由分区数和并行化因子决定。因此，请根据吞吐量要求对其进行调整。
- 考虑查看用于批量处理的流数据解决方案白皮书[8]、分析流和其他有用的模式。
- 不使用 KPL 时，请确保考虑非原子操作（如 PutRecords）的部分故障，因为 Kinesis API 会在提取时返回成功处理和未成功处理的记录[9]。
- 可能会出现重复记录[10]，您必须在应用程序中为消费者和生产者利用重试和幂等性这两种功能。
- 当提取的数据需要持续加载到 Amazon S3、Amazon Redshift 或 Amazon ES 中时，考虑使用 Lambda 上的 Kinesis Data Firebose。
- 可以使用标准 SQL 查询流数据时，考虑使用 Kinesis Data Analytics，而非 Lambda，并只将查询结果加载到 Amazon S3、Amazon Redshift、Amazon ES 或 Kinesis Streams。
- 遵循基于 AWS Lambda 流的调用[11]的最佳实践，因为其中更详细地介绍了对批处理大小、每个分区的并发性和监控流处理的影响。
- 使用 Lambda 的最大重试次数、最长记录存留期、对函数错误进行等分批处理和失败时目标错误控制构建更具弹性的流处理应用程序。

Web 应用程序

Web 应用程序通常具有严格要求，以确保一致、安全、可靠的用户体验。为确保高可用性、全球可用性，以及扩展到数千或潜在数百万用户的能力，您通常必须预留大量的多余容量，以应对 Web 请求的最高预期需求。这通常需要管理大量服务器和额外的基础设施组件，进而会显著增加资本支出并延长容量预置准备期。

使用 AWS 上的无服务器计算，您可以部署整个 Web 应用程序堆栈，而无需执行管理服务器、预估预置容量或为闲置资源付费等千篇一律的繁重工作。此外，您也不必折损安全性、可靠性或性能。

特征：

- 您需要可在数分钟内实现全球化部署并具有高水平的弹性和可用性的可扩展 Web 应用程序。
- 您需要始终如一且响应足够快速的用户体验。
- 您希望尽可能地在平台中利用托管服务，以限制与管理通用平台相关的繁重工作。
- 您希望根据实际用户需求优化成本，而不是为闲置资源付费。
- 您希望创建一个易于设置和操作且在以后进行扩展时产生有限影响的框架。

参考架构

图 6：Web 应用程序参考架构

1. 此 Web 应用程序的**使用者**可能在地域上集中分布，也可能遍布世界各地。利用 Amazon CloudFront 不仅可以通过缓存和最优的源路由为这些使用者提供更好的性能体验，还可以限制对后端的冗余调用。
2. **Amazon S3** 托管 Web 应用程序静态资产，并通过 CloudFront 获得安全保护。
3. **Amazon Cognito 用户池**为 Web 应用程序提供用户管理和身份提供程序功能。
4. 在许多情况下，由于使用者从 Amazon S3 下载静态内容，因此需要向您的应用程序发送动态内容或应用程序接收动态内容。例如，当用户通过表单提交数据时，**Amazon API Gateway** 充当安全终端节点，执行这些调用并返回通过 Web 应用程序显示的响应。
5. **AWS Lambda** 函数基于 DynamoDB 为您的 Web 应用程序提供创建、读取、更新和删除 (CRUD) 操作。
6. **Amazon DynamoDB** 可提供后端 NoSQL 数据存储，随着 Web 应用程序的流量弹性扩展。

配置说明：

- 遵循在 AWS 上部署无服务器 Web 应用程序前端的最佳实践。有关更多信息，请参阅卓越运营支柱。
- 对于单页 Web 应用程序，使用 AWS Amplify 控制台管理原子部署、缓存过期、自定义域和用户界面 (UI) 测试。
- 有关身份验证和授权方面的建议，请参阅安全性支柱。
- 有关 Web 应用程序后端方面的建议，请参阅 RESTful 微服务场景。
- 对于提供个性化服务的 Web 应用程序，可以利用 API Gateway 使用计划[12]和 Amazon Cognito 用户池来确定不同用户集有权访问的范围。例如，高级用户可具有更高的 API 调用吞吐量，可访问更多 API、更多存储等。
- 如果您的应用程序使用本场景中未涵盖的搜索功能，请参阅移动后端场景。

架构完善的框架的支柱

本节将介绍每个支柱，包括定义、最佳实践、问题、注意事项和为无服务器应用程序构建解决方案时涉及到的关键 AWS 服务。

为简洁起见，我们只从架构完善的框架中选择了特定于无服务器工作负载的问题。在设计架构时，本文未包含的问题也应考虑在内。我们建议阅读 AWS 架构完善的框架白皮书。

卓越运营支柱

卓越运营支柱包括运行和监控系统以创造业务价值并持续改善支持流程和程序的能力。

定义

在云中实现卓越运营有三个领域的最佳实践：

- 准备
- 运营
- 演进

除了架构完善的框架所涵盖的流程、运行手册和实际试用相关内容之外，您还应该了解一些特定领域，以在无服务器应用程序中推动实现卓越运营。

最佳实践

准备

本小节没有独特的无服务器应用程序运营实践。

运营

> 运营 1：如何了解无服务器应用程序的运行状况？

指标和警报

了解您打算使用的每个 AWS 服务的 Amazon CloudWatch 指标和维度非常重要，这样您就可以制定计划以评估其行为，并在您认为合适的地方添加自定义指标。

Amazon CloudWatch 提供自动化跨服务控制面板和基于服务的控制面板，可帮助您了解所用 AWS 服务的关键指标。对于自定义指标，可使用 Amazon CloudWatch 嵌入式指标格式记录一批指标，这些指标将由 CloudWatch 异步处理，而不会影响无服务器应用程序的性能。

无论您要创建控制面板，还是希望为新的和现有的应用程序制定计划，但凡涉及到指标，都可以使用以下准则：

- **业务指标**
    - 业务 KPI 将根据业务目标衡量应用程序的性能，对您了解严重影响整体业务的情况、是否赚取利润非常重要。
    - **示例**：下单量、借记卡/信用卡业务、购买机票量等。
- **客户体验指标**
    - 客户体验数据不仅表明了 UI/UX 的整体有效性，还表明了变更或异常是否影响应用程序特定部分的客户体验。通常情况下，这些指标以百分比进行度量，以防止在尝试了解一段时间内的影响以及对客户群的影响情况时出现异常值。
    - **示例**：可感知的延迟、将商品加入购物篮或结账的时间、页面加载时间等。

- **系统指标**
    - 供应商和应用程序指标对于证实前几节的根本原因非常重要。这些指标还会指明您的系统运行正常、存在风险，还是已是您的客户。
    - **示例**：HTTP 错误/成功百分比、内存利用率、运行时长/错误/限制、队列长度、流记录长度、集成延迟等。
- **运营指标**
    - 运营指标对于了解给定系统的可持续性和维护状况同样重要，对于查明稳定性如何随时间提升/下降也至关重要。
    - **示例**：请求单数量（成功和不成功的解决方案等）、服务人员收到呼叫的次数、可用性、CI/CD 管道统计数据（成功/失败的部署、反馈时间、周期和准备时间等等）。

CloudWatch 警报应同时在单独级别和聚合级别进行配置。以下是一个单独级别示例：针对 Lambda 的 *Duration* 指标设置警报，或通过 API 调用时，针对 API Gateway 的 *IntegrationLatency* 设置警报，因为应用程序的不同部分可能具有不同的配置文件。在此情况下，您可以快速识别出导致函数执行时间远长于常用时间的错误部署。

聚合级别示例包括但不限于针对以下指标设置警报：

- **AWS Lambda**：*Duration*、*Errors*、*Throttling* 和 *ConcurrentExecutions*。对于基于流的调用，针对 *IteratorAge* 设置警报。对于异步调用，针对 *DeadLetterErrors* 设置警报。
- **Amazon API Gateway**：*IntegrationLatency*、*Latency*、*5XXError*
- **Application Load Balancer**：*HTTPCode_ELB_5XX_Count*、*RejectedConnectionCount*、*HTTPCode_Target_5XX_Count*、*UnHealthyHostCount*、*LambdaInternalError*、*LambdaUserError*
- **AWS AppSync**：5XX 和 *Latency*
- **Amazon SQS：***ApproximateAgeOfOldestMessage*

- **Amazon Kinesis Data Streams：***ReadProvisionedThroughputExceeded*、*WriteProvisionedThroughputExceeded*、*GetRecords.IteratorAgeMilliseconds*、*PutRecord.Success*、*PutRecords.Success*（如果使用 Kinesis Producer Library）和 *GetRecords.Success*
- **Amazon SNS**：*NumberOfNotificationsFailed*、*NumberOfNotificationsFilteredOut-InvalidAttributes*
- **Amazon SES：***Rejects*、*Bounces*、*Complaints*、*Rendering Failures*
- **AWS Step Functions：***ExecutionThrottled*、*ExecutionsFailed*、*ExecutionsTimedOut*
- **Amazon EventBridge：***FailedInvocations*、*ThrottledRules*
- **Amazon S3：***5xxErrors*、*TotalRequestLatency*
- **Amazon DynamoDB：***ReadThrottleEvents*、*WriteThrottleEvents*、*SystemErrors*、*ThrottledRequests*、*UserErrors*

集中式和结构化日志记录

将应用程序日志记录标准化，以提供关于事务、相关标识符、跨组件的请求标识符和业务成果的操作信息。使用这些信息可回答有关工作负载状态的任意问题。

下面是一个使用 JSON 作为输出的结构化日志记录示例：

```
{
   "timestamp":"2019-11-26 18:17:33,774",
   "level":"INFO",
   "location":"cancel.cancel_booking:45",
   "service":"booking",
   "lambda_function_name":"test",
   "lambda_function_memory_size":"128",
   "lambda_function_arn":"arn:aws:lambda:eu-west-1:
12345678910:function:test",
   "lambda_request_id":"52fdfc07-2182-154f-163f-5f0f9a621d72",
   "cold_start": "true",
   "message":{
      "operation":"update_item",
      "details":{
         "Attributes":{
            "status":"CANCELLED"
```

```
          },
          "ResponseMetadata":{

"RequestId":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG",
             "HTTPStatusCode":200,
             "HTTPHeaders":{
                "server":"Server",
                "date":"Thu, 26 Nov 2019 18:17:33 GMT",
                "content-type":"application/x-amz-json-1.0",
                "content-length":"43",
                "connection":"keep-alive",
                "x-amzn-
requestid":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG",
                "x-amz-crc32":"1848747586"
             },
             "RetryAttempts":0
          }
       }
    }
}
```

集中式日志记录可帮助您搜索和分析无服务器应用程序日志。结构化日志记录可支持您更轻松地实现查询，回答关于应用程序运行状况的任意问题。随着系统扩展和所提取日志记录的增加，可以考虑使用适当的日志记录级别和抽样机制来在调试模式下记录一小部分日志。

分布式跟踪

与无服务器应用程序类似，分布式系统中也可能会大规模出现异常。根据无服务器架构的性质，分布式跟踪是基础设置。

对无服务器应用程序进行更改需要许多与传统工作负载中使用的部署、更改和发布管理相同的原则。然而，在如何使用现有工具来实施这些原则方面有一些细微变化。

应启用 AWS X-Ray 的活动跟踪，以提供分布式跟踪功能，同时启用可视化服务映射，以便更快地排除故障。X-Ray 可帮助发现性能下降问题及快速了解异常，包括延迟分布情况。

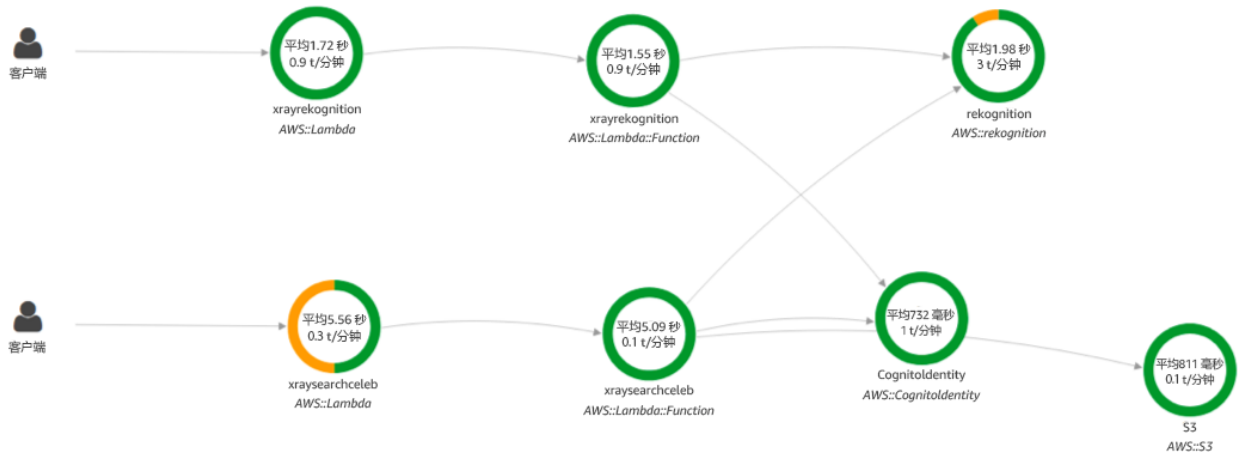

图 7：显示 2 个服务的 AWS X-Ray 服务映射

服务映射有助于了解需要注意的集成点以及弹性实践。对于集成调用，需要重试、回退且可能需要断路器，以防止故障传播到下游服务。

另一个示例是网络异常。您不应依赖默认的超时和重试设置。相反，如果在某些客户端中发生套接字读/写超时（默认值可以是秒，也可以是分钟），则对其进行调整以便快速失败。

X-Ray 还提供两项强大功能，这些功能可以提高发现应用程序异常的效率：注释和子分段。

子分段有助于了解应用程序逻辑的构造原理及其必须与之通信的外部依赖项。注释是具有字符串、数字或布尔值的键值对，由 AWS X-Ray 自动编制索引。

两者相结合，可以帮助您快速识别特定操作和业务事务的性能统计数据，例如，查询数据库所用的时长，或者处理拥有大量人群的图片所用的时长。

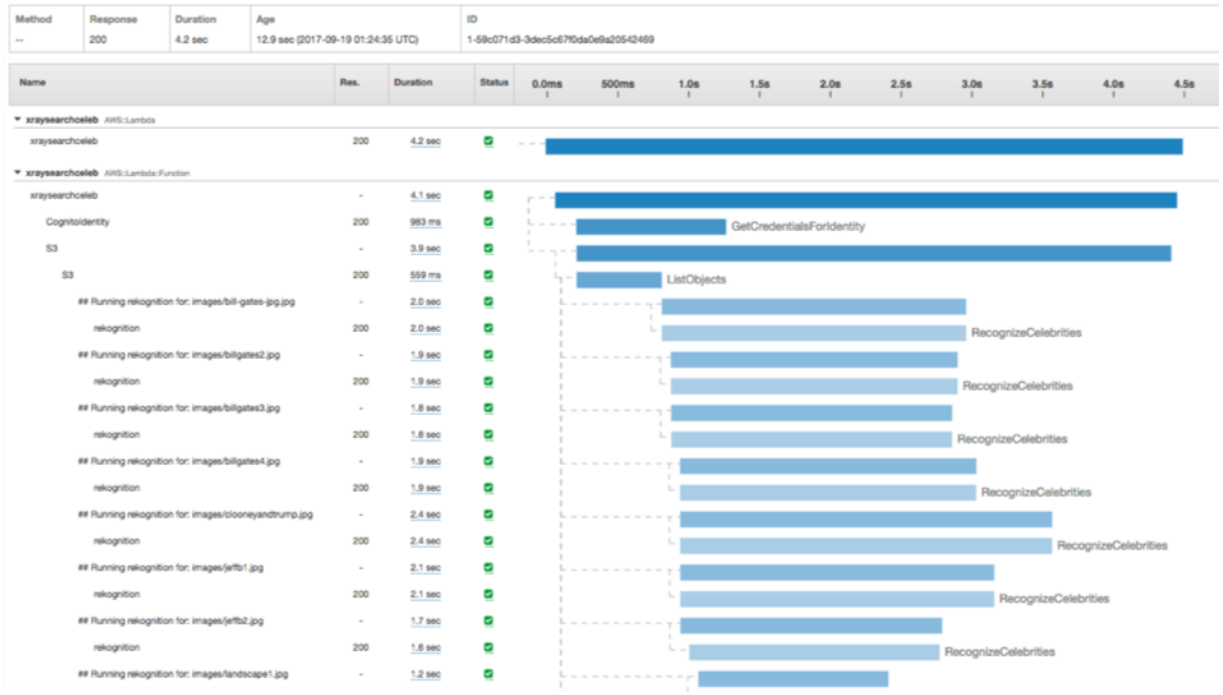

图 8：AWS X-Ray 跟踪，子分段以 ## 开头

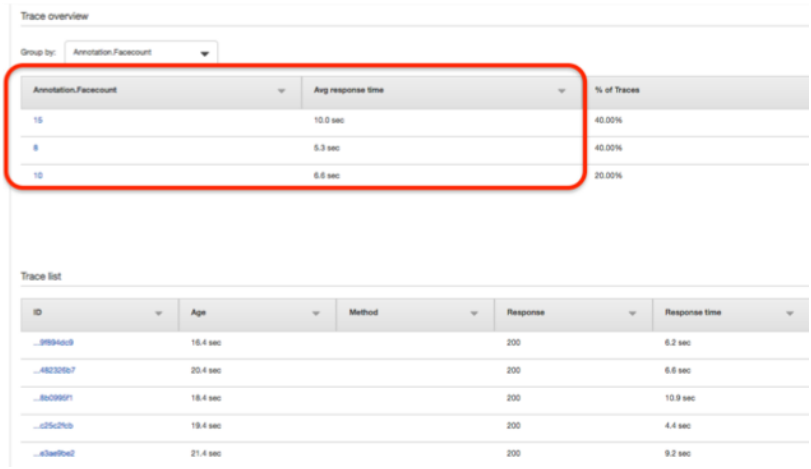

图 9：按自定义注释分组的 AWS X-Ray 跟踪

运营 2：如何执行应用程序生命周期管理？

原型设计

使用基础设施即代码为要设计原型的新功能创建临时环境，并在完成后予以清理。根据团队规模和组织内的自动化水平，您可以为每个团队或每个开发人员使用专用账户。

临时环境可帮助您在使用托管服务时实现更高的保真度，并提高控制级别，以便确保工作负载按预期集成和运行。

对于配置管理，可使用环境变量进行不频繁的更改，例如日志记录级别和数据库连接字符串。使用 AWS System Manager Parameter Store 进行动态配置（例如功能切换），并使用 AWS Secrets Manager 存储敏感数据。

测试

测试通常通过单元测试、集成测试和验收测试来完成。制定稳健的测试策略可使您在不同的负载和条件下模拟无服务器应用程序。

单元测试应与非无服务器应用程序相同，因此可以在本地运行，而不需要进行任何更改。

集成测试不应模拟您无法控制的服务，因为它们可能会更改并提供意外结果。这些测试在使用实际服务时性能更优，因为它们可以提供无服务器应用程序在生产环境中处理请求时将使用的相同环境。

验收测试或端到端测试应在不进行任何更改的情况下执行，因为其主要目标是通过可用的外部接口模拟最终用户的操作。因此，此类测试没有需要注意的独特建议。

通常，Lambda 和 AWS Marketplace 中提供的第三方工具可以在性能测试环境中用作测试用具。以下是在进行性能测试时需要注意的一些事项：

- CloudWatch Logs 中提供调用的最大已用内存和初始化持续时间等指标。有关更多信息，请参阅性能支柱章节。
- 如果您的 Lambda 函数在 Amazon Virtual Private Cloud (VPC) 内运行，请注意子网中的可用 IP 地址空间。
- 将模块化代码作为处理程序外部的独立函数创建可以实现更多的单元可测试函数。
- 建立在 Lambda 函数的静态构造函数/初始化代码（即，处理程序外部的全局范围）中引用的外部化连接代码（例如关系数据库的连接池）将确保在重用 Lambda 执行环境时不会达到外部连接阈值。
- 除非性能测试超出您账户中的当前限制，否则使用 DynamoDB 按需表。
- 考虑执行性能测试的无服务器应用程序中可能使用的任何其他服务限制。

部署

使用基础设施即代码和版本控制来跟踪更改和发布。将开发阶段和生产阶段隔离在单独的环境中。这样可减少手动流程引起的错误，并有助于加强控制，以便确保工作负载按预期运行。

使用无服务器框架（例如 AWS SAM 或 Serverless Framework）对无服务器应用程序进行建模、原型设计、构建、打包和部署。利用基础设施即代码和框架，您可以对无服务器应用程序及其依赖项进行参数化，从而简化跨隔离阶段和跨 AWS 账户的部署。

例如，CI/CD 管道 Beta 阶段会在 Beta AWS 账户中创建以下资源，同样，相应的阶段会在不同账户（Gamma、Dev、Prod）中创建相应的资源：*OrderAPIBeta*、*OrderServiceBeta*、*OrderStateMachineBeta*、*OrderBucketBeta*、*OrderTableBeta*。

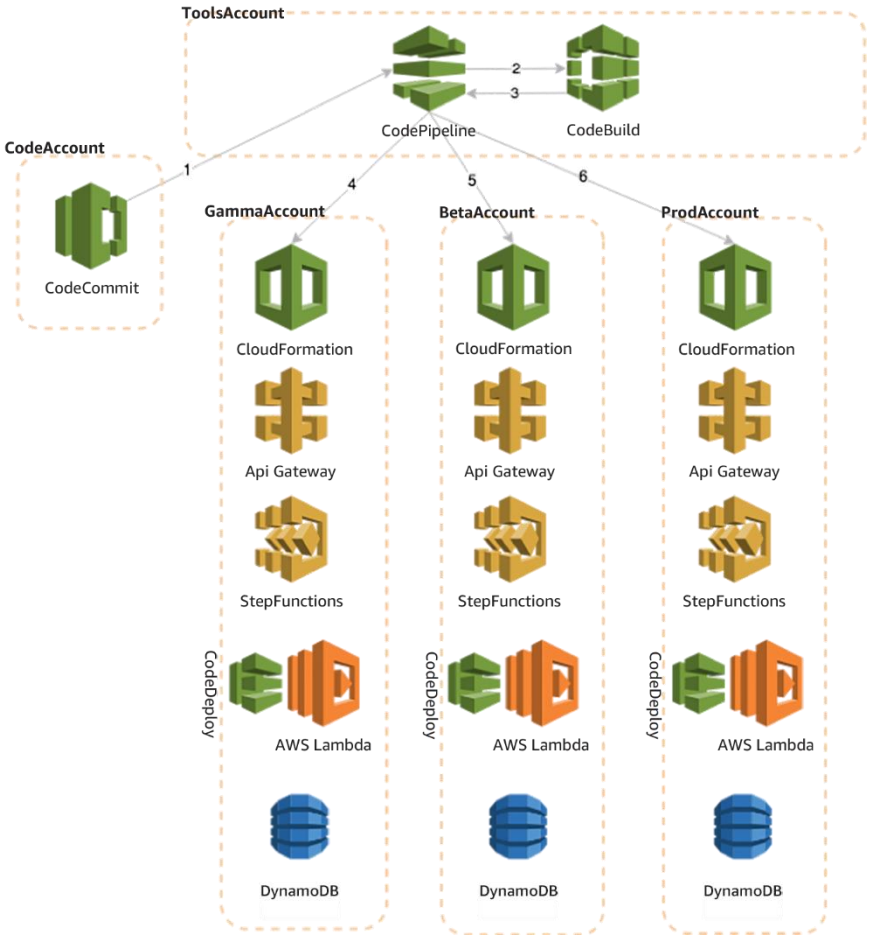

图 10：用于多个账户的 CI/CD 管道

部署到生产环境时，请优先考虑安全部署，而不是一次性部署所有系统，因为新的更改将随着时间的推移逐步转向 Canary 或线性部署中的最终用户。使用 CodeDeploy 挂钩（*BeforeAllowTraffic*、*AfterAllowTraffic*）和警报，以便更好地控制部署验证、回滚和应用程序可能需要的任何自定义。

您还可以在部署过程中结合使用合成流量、自定义指标和警报。这些可以帮助您主动检测新更改中的错误，以防影响您的客户体验。

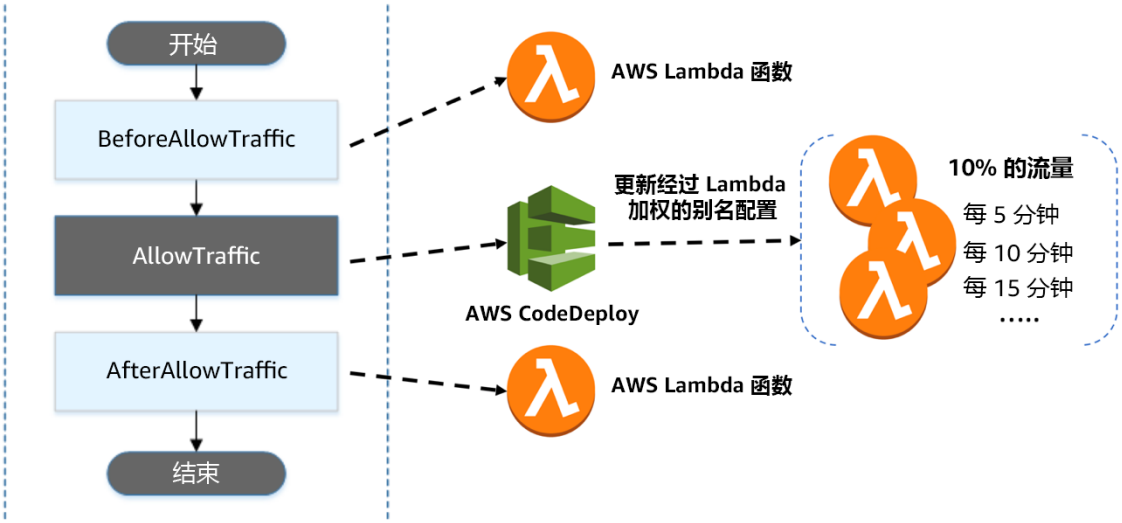

图 11：AWS CodeDeploy Lambda 部署和挂钩

演进

本小节没有独特的无服务器应用程序运营实践。

关键 AWS 服务

实现卓越运营的关键 AWS 服务包括 AWS Systems Manager Parameter Store、AWS SAM、CloudWatch、AWS CodePipeline、AWS X-Ray、Lambda 和 API Gateway。

资源

请参阅以下资源，详细了解卓越运营的最佳实践。

文档和博客

- API Gateway 阶段变量[13]
- Lambda 环境变量[14]
- AWS SAM CLI[15]
- X-Ray 延迟分配[16]
- 使用 X-Ray 排查基于 Lambda 的应用程序的故障[17]
- System Manager (SSM) Parameter Store[18]
- 持续部署无服务器应用程序博客文章[19]
- SamFarm：CI/CD 示例[20]
- 使用 CI/CD 的无服务器应用程序示例
- 自动设置警报和控制面板的无服务器应用程序示例
- 适用于 Python 的 CloudWatch 嵌入式指标格式库
- 适用于 Node.js 的 CloudWatch 嵌入式指标格式库
- 实施跟踪、结构化日志记录和自定义指标的示例库
- 一般 AWS 限制
- Stackery：多账户最佳实践

白皮书

- 在 AWS 上实践持续集成/持续交付[21]

第三方工具

- 无服务器开发人员工具页面，包括第三方框架/工具[22]
- Stelligent：提供运营指标的 CodePipeline 控制面板

安全性支柱

安全性支柱包括通过风险评估和缓解策略在提供业务价值的同时保护信息、系统和资产的能力。

定义

在云中实现安全性有五个最佳实践领域：

- 身份与访问管理
- 检测性控制
- 基础设施保护
- 数据保护
- 事件响应

无服务器可消除执行修补操作系统、更新二进制文件等基础设施管理任务的需要，因而可解决现今最大的某些安全隐患。尽管与非无服务器架构相比攻击面有所减小，但开放式 Web 应用程序安全项目 (OWASP) 和应用程序安全性最佳实践仍然适用。

本节中的问题旨在帮助您应对攻击者试图获取访问权限或利用错误配置的权限的特定途径，这些途径可能会导致滥用。本节所述的实践对整个云平台的安全性都有很大影响，因此应该仔细验证并经常审核这些实践。

本文档不包含对**事件响应**类别的介绍，因为 AWS 架构完善的框架中的实践仍然适用。

最佳实践

身份与访问管理

> 安全性 1：如何控制对无服务器 API 的访问？

由于 API 可以执行操作并获得有价值的数据，因此 API 经常成为攻击者的目标。抵御这些攻击的安全最佳实践多种多样。

从身份验证/授权的角度来看，目前可通过四种机制在 API Gateway 内授权 API 调用：

- AWS_IAM 授权
- Amazon Cognito 用户池

- API Gateway Lambda 授权方
- 资源策略

首先，您需要了解这些机制是否已实施以及如何实施。对于当前位于您的 AWS 环境内或有办法检索 AWS Identity and Access Management (IAM) 临时凭证来访问您的环境的使用者，可以通过使用 AWS_IAM 授权并向相应的 IAM 角色添加最低权限来安全地调用您的 API。

下图说明了在这种情况下如何使用 AWS_IAM 授权：

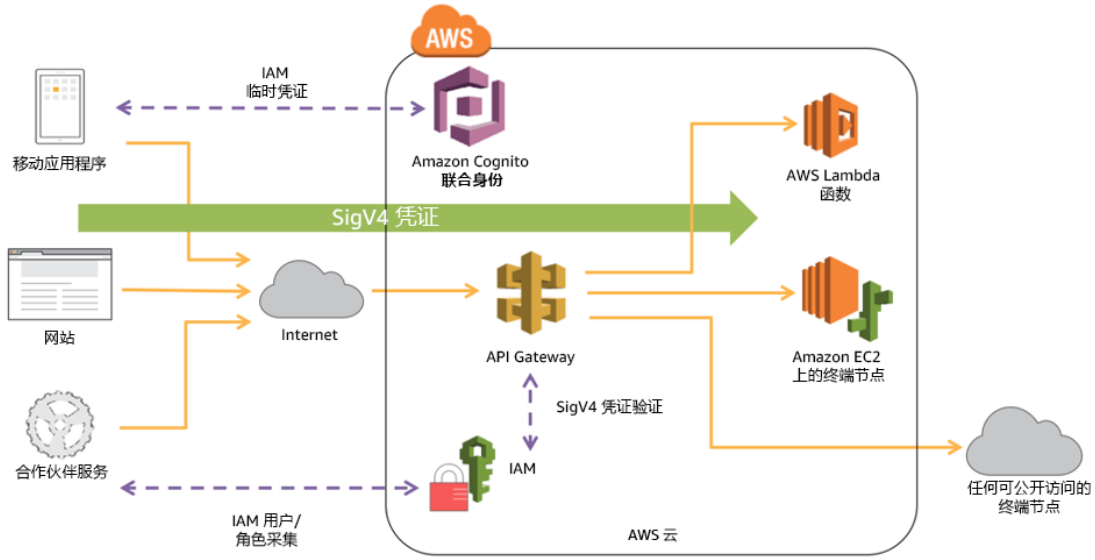

图 12：AWS_IAM 授权

如果您已有身份提供商 (IdP)，可以使用 API Gateway Lambda 授权方来调用 Lambda 函数以根据您的 IdP 对给定用户进行身份验证/验证。您可以根据身份元数据将 Lambda 授权方用于自定义验证逻辑。

Lambda 授权方可将持有者令牌或请求上下文值中派生出的附加信息发送到您的后端服务。例如，授权方可返回包含用户 ID、用户名和范围的映射。使用 Lambda 授权方后，您的后端无需将授权令牌映射到以用户为中心的数据，因此您可以仅向授权函数公开此类信息。

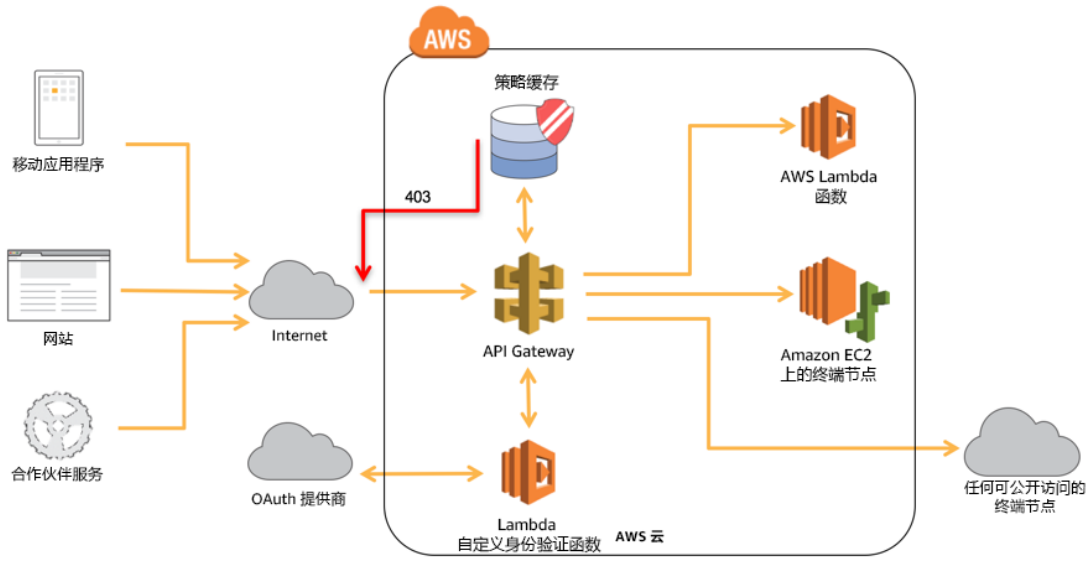

图 13：API Gateway Lambda 授权方

如果您没有 IdP，可以利用 Amazon Cognito 用户池来提供内置用户管理或与外部身份提供商（例如 Facebook、Twitter、Google+ 和 Amazon）集成。

这种方法在移动后端场景中很常见，在此类场景中，用户可以使用社交媒体平台中的现有账户进行身份验证，同时也可使用自己的电子邮件地址/用户名进行注册/登录。这种方法还通过 Oauth 范围提供精细的授权。

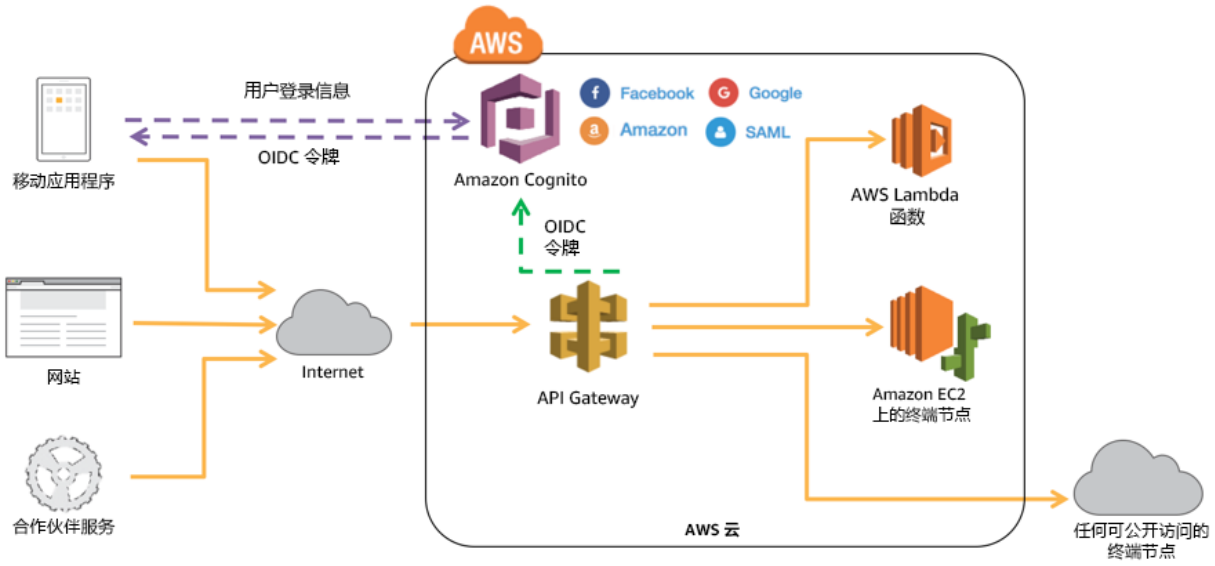

图 14：Amazon Cognito 用户池

API Gateway API 密钥不属于安全机制。除非它是公有 API，否则不应将其用于授权。它主要用于跟踪使用者对您的 API 的使用情况，并可用作本节前面提到的授权方的补充。

使用 Lambda 授权方时，我们严格建议不要通过查询字符串参数或标题来传递凭证或任何类型的敏感数据，否则您的系统可能会招至滥用。

Amazon API Gateway 资源策略是 JSON 策略文档，您可以将其附加到 API 来控制指定的 AWS 委托人能否调用 API。

此机制可让您通过以下方式限制 API 的调用：

- 指定的 AWS 账户或任何 AWS IAM 身份中的用户
- 指定的源 IP 地址范围或 CIDR 块
- 指定的 Virtual Private Cloud (VPC) 或 VPC 终端节点（在任何账户中）

使用资源策略，您可以限制常见场景，例如只允许来自具有特定 IP 范围的已知客户端或来自其他 AWS 账户的请求。如果您计划限制来自私有 IP 地址的请求，建议使用 API Gateway 私有终端节点。

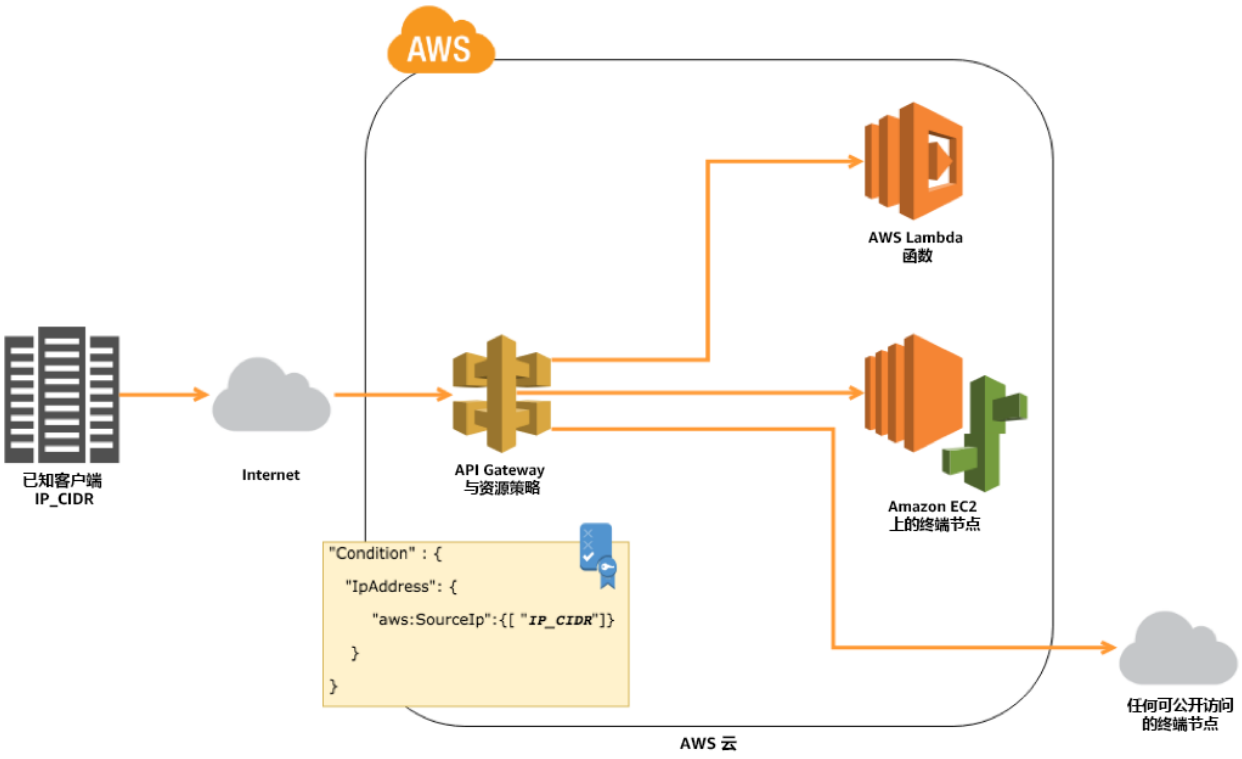

图 15：基于 IP CIDR 的 Amazon API Gateway 资源策略

借助私有终端节点，API Gateway 将限制对您 VPC 中服务和资源的访问，或限制通过 Direct Connect 连接到您自己的数据中心的服务和资源的访问。

通过结合使用私有终端节点和资源策略，可将 API 限制为特定私有 IP 范围内的特定资源调用。此种结合主要用于可能位于同一账户或其他账户的内部微服务。

当涉及到大型部署和多个 AWS 账户时，组织可以利用 API Gateway 中跨账户的 Lambda 授权方来减少维护工作和集中处理安全实践。例如，API Gateway 可以在单独的账户中使用 Amazon Cognito 用户池。Lambda 授权方也可以在单独的账户中进行创建和管理，然后重新用于 API Gateway 管理的多个 API。对于需要跨 API 标准化授权实践的多个微服务的部署，这两个场景都很常见。

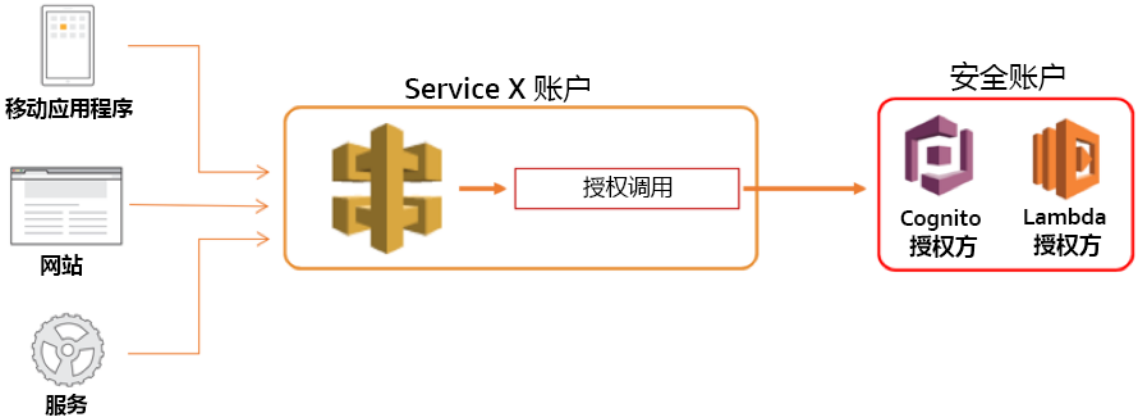

图 16：API Gateway 跨账户授权方

安全性 2：如何管理无服务器应用程序的安全边界？

对于 Lambda 函数，建议您遵循最低访问权限原则，并仅授予执行给定操作所需的访问权限。为角色附加超出需求的权限，可能会导致系统遭到滥用。

对于安全上下文，使用执行限定范围的活动的较小函数有助于实现架构更完善的无服务器应用程序。对于 IAM 角色，在多个 Lambda 函数中共享 IAM 角色很可能会违反最低访问权限原则。

检测性控制

由于安全/取证、法律或法规要求等原因，日志管理成为架构良好的设计的重要组成部分。

在应用程序依赖项内跟踪漏洞同样也很重要，因为无论使用哪种编程语言，攻击者都能利用在依赖项中发现的已知漏洞。

对于应用程序依赖项漏洞扫描，可以在您的 CI/CD 管道中集成多个商用和开源解决方案（例如 OWASP 依赖关系检查）。务必要将所有依赖项（包括 AWS 开发工具包）添加为版本控制软件存储库的一部分。

基础设施保护

对于您的无服务器应用程序需要与其他部署在 Virtual Private Cloud (VPC) 中的组件或驻留在本地的应用程序进行交互的场景，务必要考虑网络边界。

可对 Lambda 函数进行配置以访问 VPC 中的资源。按照 AWS 架构完善的框架中的描述，控制所有层中的流量。对于由于合规性原因而需要进行出站流量筛选的工作负载，可以用在非无服务器架构中使用代理的方式使用代理。

由于关注点分离，不建议仅在应用程序代码级别强制执行网络边界并给出关于可以访问哪些资源的指示。

对于服务间通信，应偏重动态身份验证，例如使用 AWS IAM 的临时凭证而非静态密钥。API Gateway 和 AWS AppSync 都支持 IAM 授权，该授权是保护与 AWS 服务之间的通信的理想方法。

数据保护

考虑启用 API Gateway 访问日志，并选择性地选择所需内容，因为日志可能包含敏感数据，具体取决于您的无服务器应用程序设计。出于此原因，我们建议您加密遍历您的无服务器应用程序的任何敏感数据。

API Gateway 和 AWS AppSync 在所有通信、客户端和集成中使用 TLS。尽管 HTTP 有效负载已在传输过程中加密，但 URL 中的请求路径和查询字符串可能未加密。因此，如果将敏感数据发送到标准输出，则可能会通过 CloudWatch Logs 意外泄露这些数据。

此外，格式错误或拦截的输入会被用作攻击媒介以获取系统访问权限或导致故障。应按照 AWS 架构完善的框架中的详细论述，保护所有可能的层中的敏感数据。在此情景中，该白皮书中的建议仍然适用。

对于 API 网关，敏感数据应先在客户端进行加密，然后作为 HTTP 请求的一部分发送，或者作为 HTTP POST 请求的一部分作为有效负载发送。这还包括在发出给定请求之前，对任何可能包含敏感数据的标头进行加密。

对于 Lambda 函数或 API Gateway 可能配置的任何集成，在进行任何处理和数据操作之前都应加密敏感数据。如果此类数据在持久性存储中公开或通过 CloudWatch Logs 流式传输和持久保存的标准输出公开，这将能防止数据泄露。

在本文档前面描述的场景中，Lambda 函数会将加密的数据保存在 DynamoDB、Amazon ES 或 Amazon S3 中，同时还会进行静态加密。我们强烈建议不要发送、记录和存储未加密的敏感数据，不管是作为 HTTP 请求路径/查询字符串的组成部分，还是位于 Lambda 函数的标准输出中的数据。

也不建议在敏感数据未加密的 API Gateway 中启用日志记录。如检测性控制小节所述，在这种情况下，启用 API Gateway 日志记录之前，您应该先咨询您的合规性团队。

> 安全性 3：如何在工作负载中实现应用程序安全性？

查看由 AWS 架构完善的框架中介绍的 AWS 安全公告和行业威胁情报编写的安全意识文档。适用于应用程序安全性的 OWASP 原则仍然适用。

验证和清理入站事件，并像通常对非无服务器应用程序执行的操作一样来执行安全代码审查。对于 API Gateway，第一步是设置基本请求验证，以确保请求符合配置的 JSON-Schema 请求模型以及 URI、查询字符串或标头中的任何所需参数。应实施特定于应用程序的深层验证，无论该验证是作为单独的 Lambda 函数、库、框架还是服务。

将您的密钥（例如数据库密码或 API 密钥）存储在 Secrets Manager 中，Secrets Manager 支持轮换密钥、保证安全和进行经审计的访问。Secrets Manager 允许对密钥执行精细策略，包括审计。

关键 AWS 服务

确保安全性所需的关键 AWS 服务包括 Amazon Cognito、IAM、Lambda、CloudWatch Logs、AWS CloudTrail、AWS CodePipeline、Amazon S3、Amazon ES、Amazon DynamoDB 和 Amazon Virtual Private Cloud (Amazon VPC)。

资源

请参阅以下资源，详细了解安全性的最佳实践。

文档和博客

- 以 Amazon S3 为例说明 Lambda 函数的 IAM 角色[23]
- 验证 API Gateway 请求[24]
- API Gateway Lambda 授权方[25]
- Securing API Access with Amazon Cognito Federated Identities, Amazon Cognito User Pools, and Amazon API Gateway[26]
- 为 AWS Lambda 配置 VPC 访问[27]
- 使用 Squid Proxies 筛选 VPC 出站流量[28]
- Using AWS Secrets Manager with Lambda
- 使用 AWS Secrets Manager 审计密钥
- OWASP 输入验证速查表
- AWS 无服务器安全性研讨会

白皮书

- OWASP 安全编码最佳实践[29]
- AWS 安全最佳实践[30]

合作伙伴解决方案

- PureSec 无服务器安全性
- Twistlock 无服务器安全性[31]
- Protego 无服务器安全性
- Snyk – 商业漏洞数据库和依赖项检查[32]
- 结合使用 Hashicorp Vault 与 Lambda 和 API Gateway

第三方工具

- OWASP 漏洞依赖项检查[33]

可靠性支柱

可靠性支柱包含系统从基础设施中断或服务中断恢复、动态获取计算资源以满足需求以及减少中断（如错误配置或暂时性网络问题）的能力。

定义

在云中有三个领域的可靠性最佳实践：

- 基础
- 变更管理
- 故障管理

为实现可靠性，系统必须具有经过周密规划的基础和适当的监控功能，同时能够根据需求、具体要求处理各项变更，或防御可能出现的未经授权的拒绝服务攻击。理想情况下，系统的设计应具有故障检测和自动修复功能。

最佳实践

基础

> 可靠性 1：您如何管理入站请求速率？

限流

在微服务架构中，API 使用者可能在单独的团队中，甚至是组织外部。由于访问模式未知及使用者凭证面临被损坏的风险，这会产生漏洞。如果请求的数量超过处理逻辑/后端能够处理的数量，服务 API 可能会受到影响。

此外，触发新事务的事件（例如数据库行更新或添加到 S3 存储桶作为 API 组成部分的新对象）将在整个无服务器应用程序中触发额外的执行。

应在 API 级别启用限流，以强制执行由服务合同确定的访问模式。定义请求访问模式策略对于确定使用者如何使用服务（无论是在资源级别还是全局级别）具有重要意义。

在您的 API 中返回相关的 HTTP 状态代码（如用于限流的 429），可以帮助使用者通过相应地实施回退和重试规划受限制的访问。

对于更精细的限流和计量使用情况，除了使用全局限流外，还可以通过使用情况计划向使用者发出 API 密钥，从而使 API Gateway 能够在出现意外行为时强制执行配额和访问模式。API 密钥还可简化管理员在使用者发出可疑请求时切断访问的流程。

捕获 API 密钥的常用方法是通过开发人员门户进行捕获。这可为作为服务提供商的您提供更多与使用者和请求相关的元数据。您可以捕获应用程序、联系信息和业务领域/用途并将这些数据存储在持久性数据存储（例如 DynamoDB）中。这可让您对使用者进行额外验证，并使用身份追溯日志记录，以便您在解决更改升级/问题时联系使用者。

如安全性支柱中所述，API 密钥不是用于授权请求的安全机制，因此仅应与 API Gateway 中提供的可用授权选项之一结合使用。

有时，并发性控制对于保护特定工作负载不受服务故障影响很有必要，因为它们的扩展速度可能不及 Lambda。使用并发性控制，您可以控制在单个 Lambda 函数级别上设置特定 Lambda 函数的并发调用次数的分配。

超出单个函数组并发性的 Lambda 调用将受到 AWS Lambda 服务的限流，其结果将因其事件源的不同而异：同步调用返回 HTTP 429 错误，异步调用将进行排队并重试，而基于流的事件源会在记录过期时间前一直重试。

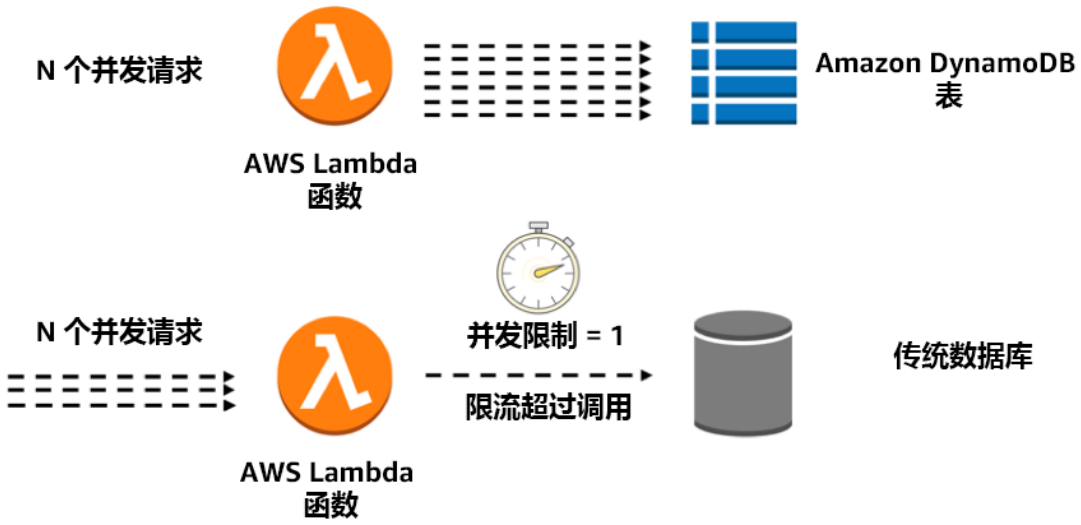

图 17：AWS Lambda 并发性控制

控制并发性尤其适用于以下场景：

- 敏感的后端或可能有扩展限制的集成系统
- 数据库连接池限制，例如可能施加并发限制的关系数据库
- 关键路径服务：应对同一个账户中的限制的优先级较高的 Lambda 函数（例如授权）与优先级较低的函数（例如后台）
- 在发生异常时能够禁用 Lambda 函数（并发性 = 0）。
- 限制所需的并行执行，以防御分布式拒绝服务 (DDoS) 攻击

Lambda 函数的并发性控制还会限制其扩展到并发集之外，并从您的账户保留的并发池中提取的能力。对于异步处理，与 Lambda 函数并发性控制相反，需要使用 Kinesis Data Streams 通过单个分区有效控制并发性。这让您能够灵活地增加分区数或并行化因子来提高您的 Lambda 函数的并发性。

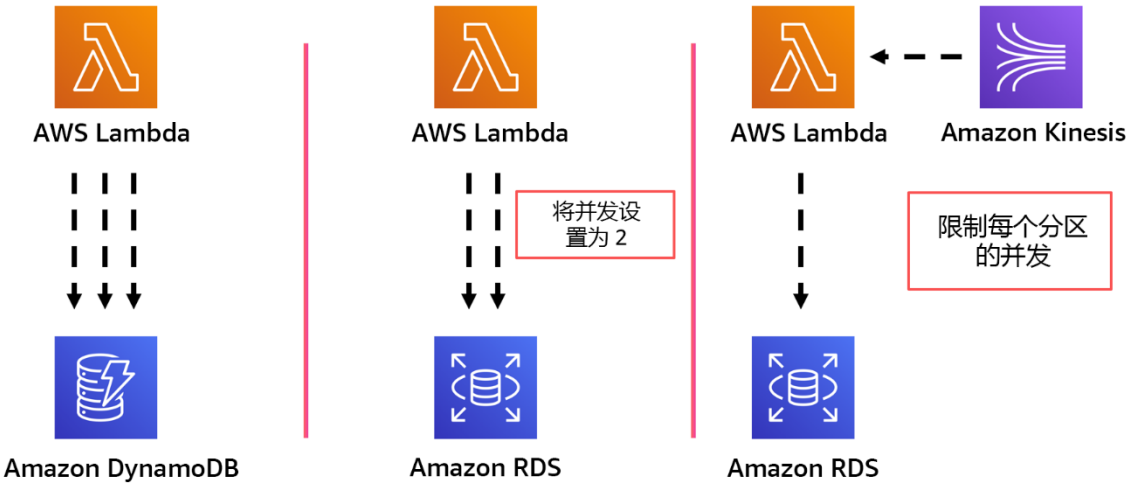

图 18：同步和异步请求的并发性控制

可靠性 2：如何将弹性构建到您的无服务器应用程序中？

异步调用和事件

异步调用可降低 HTTP 响应延迟。多个同步调用以及长时间运行的等待周期可能会导致超时和防止重试逻辑的“锁定”代码。

事件驱动的架构可以简化异步代码执行，从而限制使用者的等待周期。这些架构通常在执行业务功能的多个组件中使用队列、流、发布/订阅、Webhook、状态机和事件规则管理器异步实现。

用户体验与异步调用分离。前端系统不会屏蔽整个体验直至完成整个执行，而是会在其初始请求中接收参考/作业 ID，并且会订阅实时变更，或在旧版系统中使用其他 API 来轮询其状态。此类分离使前端能够通过使用事件循环、并行或并发技术来提高效率，同时在响应部分或完全可用的情况下发出此类请求并延迟加载应用程序的某些部件。

前端成为异步调用中的关键要素，因为它通过自定义重试和缓存变得更强大。如果由于异常、瞬时状况、网络连接或环境恶化而未收到任何响应，它可能会在可接受的 SLA 内停止正在进行的请求。

另外，如有必要进行同步调用，建议至少确保总执行时间不超过 API Gateway 或 AWS AppSync 的最大超时时间。使用外部服务（例如 AWS Step Functions）协调多个服务之间的业务事务，以控制状态并处理在请求生命周期中发生的错误处理。

变更管理

AWS 架构完善的框架对此进行了介绍，关于无服务器的具体信息请参阅卓越运营支柱。

故障管理

无服务器应用程序的某些部分由以事件驱动方式（例如通过发布/订阅以及其他模式）对各种组件的异步调用决定。当异步调用失败时，应尽可能捕获并重试它们。否则，可能会发生数据丢失，从而导致客户体验变差。

对于 Lambda 函数，请在 Lambda 查询中构建重试逻辑，以确保峰值工作负载不会使您的后端不堪重负。使用卓越运营支柱中所述的结构化日志记录记录重试，包括关于错误的上下文信息，因为它们可以作为自定义指标进行捕获。使用 Lambda 目标将有关错误、堆栈跟踪和重试的上下文信息发送至 SNS 主题和 SQS 队列等专用的死信队列 (DLQ)。您还需要制定一个计划，通过单独的机制进行轮询，以重新驱动这些失败事件，使其返回预期服务。

在与可满足大多数案例需要的其他 AWS 服务进行通信时，AWS 开发工具包会默认提供回退和重试机制。但是，请审核并调整这些机制以适应您的需求，尤其是 HTTP 保持连接、连接和套接字超时。

尽可能使用 Step Functions 来最大限度地减少无服务器应用程序中自定义尝试/捕获、回退和重试逻辑的数量。有关更多信息，请参阅成本优化支柱章节。使用 Step Functions 集成将失败的状态执行及其状态保存到 DLQ 中。

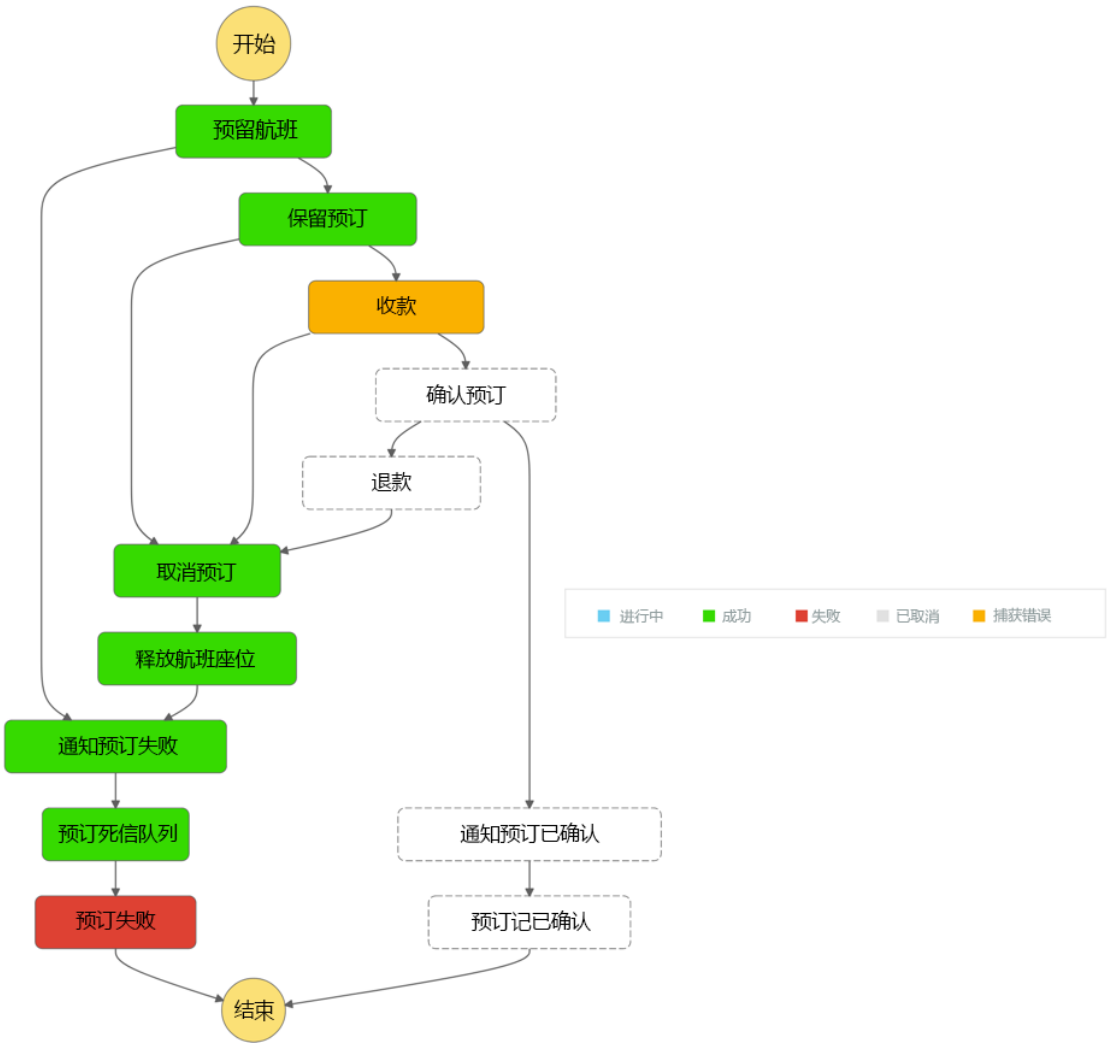

图 19：带有 DLQ 步骤的 Step Functions 状态机

在非原子操作（例如 PutRecords [Kinesis] 和 BatchWriteItem [DynamoDB]）中可能会发生局部故障，因为只要成功提取了至少一条记录，它们就会返回成功。使用此类操作时，请始终检查响应并以编程方式处理局部故障。

使用 Kinesis 或 DynamoDB 流时，请使用 Lambda 错误处理控件（例如最长记录保留期、最大重试次数、失败时的 DLQ 以及发生函数错误时的等分批处理），以便在应用程序中构建额外的弹性。

对于基于事务且依赖于某些保证和要求的同步部分，还可通过使用 Step Functions 状态机来实现 Saga 模式[34]所描述的回滚失败事务，这将解耦并简化应用程序的逻辑。

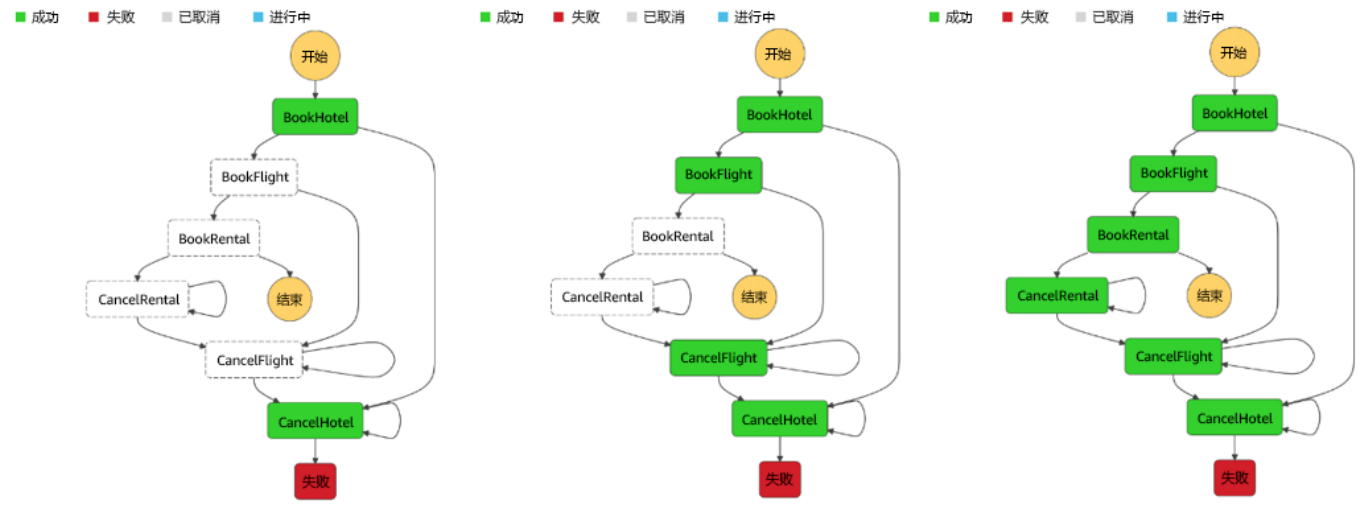

图 20：Step Functions 中的 Saga 模式，作者 Yan Cui

限制

除了架构完善的框架所述的内容之外，还可考虑查看对突发和峰值使用案例的限制。例如，API Gateway 和 Lambda 对于稳态和突发请求速率有着不同的限制。尽可能使用扩展层和异步模式并执行负载测试，以确保您的当前账户限制可以满足实际客户需求。

关键 AWS 服务

确保可靠性所需的关键 AWS 服务包括 AWS Marketplace、Trusted Advisor、CloudWatch Logs、CloudWatch、API Gateway、Lambda、X-Ray、Step Functions、Amazon SQS 和 Amazon SNS。

资源

请参阅以下资源，详细了解可靠性的最佳实践。

文档和博客

- Lambda 限制[35]
- API Gateway 限制[36]
- Kinesis Streams 限制[37]
- DynamoDB 限制[38]
- Step Functions 限制[39]
- Error handling patterns[40]
- Serverless testing with Lambda[41]
- 监控 Lambda 函数日志[42]
- 版本控制 Lambda[43]
- API Gateway 的阶段[44]
- AWS 中的 API 重试[45]
- Step Functions 错误处理[46]
- X-Ray[47]
- Lambda DLQ[48]
- Error handling patterns with API Gateway and Lambda[49]
- Step Functions 等待状态[50]
- Saga 模式[51]
- 通过 Step Functions 应用 Saga 模式[52]
- Serverless Application Repository 应用程序 – DLQ Redriver
- 对使用 AWS 开发工具包时出现的重试和超时问题进行故障排除

- Lambda resiliency controls for stream processing
- Lambda 目标
- Serverless Application Repository 应用程序 – 事件重放
- Serverless Application Repository 应用程序 – 事件存储和备份

白皮书

- AWS 上的微服务[53]

性能效率支柱

性能效率支柱专注于高效利用计算资源来满足需求，以及在需求发生变化和技术不断演进的情况下保持这种效率。

定义

云中的性能效率包含四个方面：

- 选择
- 审核
- 监控
- 权衡

采用数据驱动型方法来选择高性能架构。收集架构各方面的数据，从总体设计到资源类型的选择与配置都包括在内。通过周期性审查您的选择，您可以确保自身充分发挥 AWS 云平台持续演进所带来的优势。

监控可以确保您随时发现与预期性能的任何偏差，并采取针对性措施。最后，您可以对架构作出权衡以便提高性能，例如使用压缩或缓存，或放宽一致性要求。

> 性能 1：如何优化无服务器应用程序的性能？

选择

使用稳定和突发速率在无服务器应用程序上运行性能测试。使用相应结果，在变更后尝试调整容量单位并进行负载测试，以帮助您选择最佳配置：

- **Lambda：**测试不同的内存设置，因为 CPU、网络和存储 IOPS 是按比例分配的。
- **API Gateway：**对分散各地的客户使用边缘终端节点。对于区域客户以及在同一区域内使用其他 AWS 服务时，请使用“区域”。
- **DynamoDB：**针对不可预测的应用程序流量使用按需模式，或者针对一致的流量使用预置模式。
- **Kinesis：**在有多名使用者的场景中，对每个使用者的专用输入/输出通道使用增强扇出。通过 Lambda 对低容量事务使用扩展批处理窗口。

仅在必要时配置 VPC 对 Lambda 函数的访问。如果启用了 VPC 的 Lambda 函数需要访问 Internet，则设置 NAT 网关。如架构完善的框架所述，您可以跨多个可用区配置 NAT 网关，以实现高可用性和高性能。

API Gateway 边缘优化 API 提供了一种完全托管的 CloudFront 分配，以优化分散各地的使用者的访问。系统会将 API 请求路由至距离最近的 CloudFront 存在点 (POP)，这通常可以缩短连接时间。

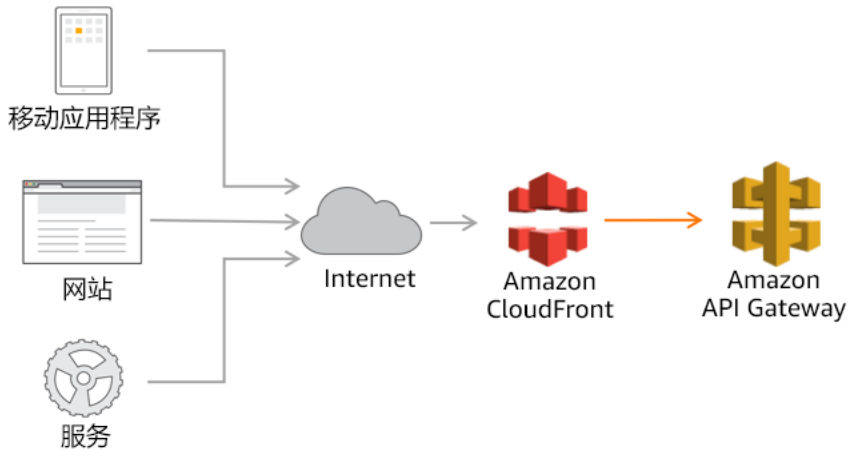

图 21：边缘优化 API Gateway 部署

API Gateway 区域终端节点默认不提供 CloudFront 分配及启用 HTTP2，这有助于在出现来自同一区域的多个请求时降低整体延迟。此外，区域终端节点还允许您关联自己的 Amazon CloudFront 分配或现有的 CDN。

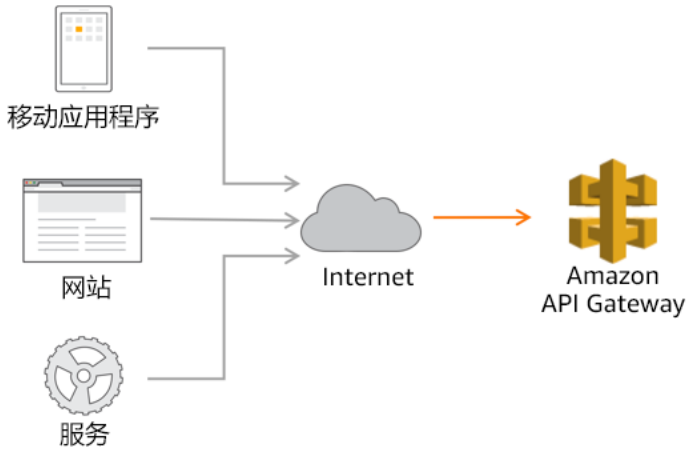

图 22：区域终端节点 API Gateway 部署

此表可帮助您决定是部署边缘优化 API 还是区域 API 终端节点：

	边缘优化 API	区域 API 终端节点
API 是跨区域访问的。包括 API Gateway 托管的 CloudFront 分配。	X	
API 是在同一区域内访问的。从部署 API 的同一区域访问 API 时的最低请求延迟。		X
能够关联自己的 CloudFront 分配。		X

此决策树可以帮助您决定何时在 VPC 中部署 Lambda 函数。

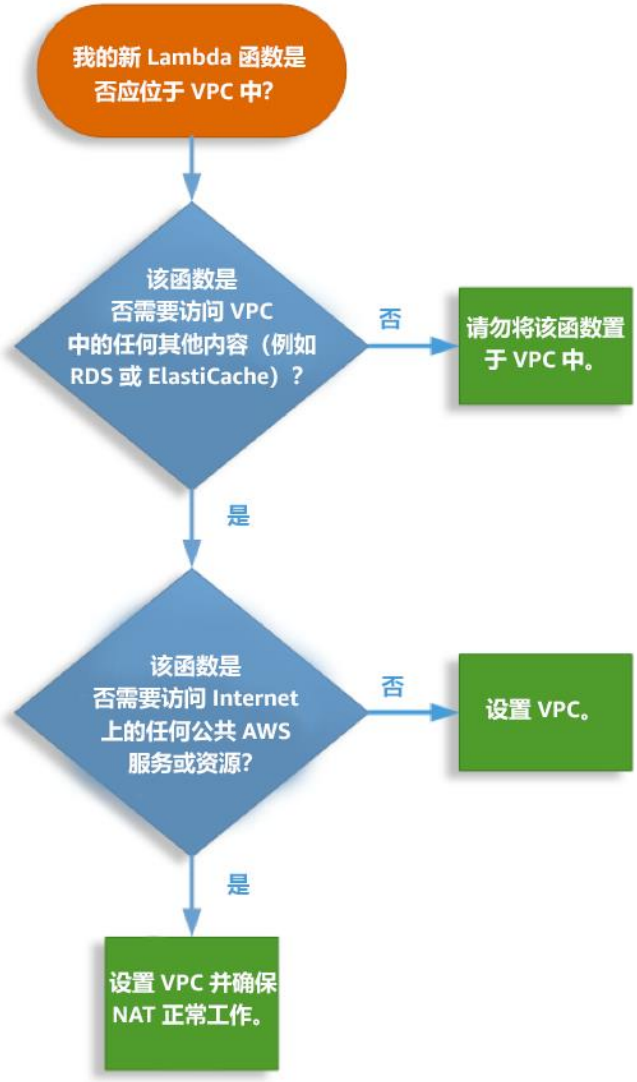

图 23：用于在 VPC 中部署 Lambda 函数的决策树

优化

随着无服务器架构的有机增长，某些机制通常跨多种工作负载配置文件使用。尽管已进行性能测试，但仍应考虑对设计进行权衡以提高应用程序的性能，同时始终牢记您的 SLA 和要求。

可以启用 API Gateway 和 AWS AppSync 缓存来提高适用操作的性能。DAX 可以显着改善读取响应以及全局和本地二级索引，从而防止 DynamoDB 全表扫描操作。这些详细信息和资源在“移动后端”场景中进行了描述。

借助 API Gateway 内容编码，API 客户端能够请求对有效负载进行压缩，然后在响应 API 请求时发回。这样可以减少从 API Gateway 发送至 API 客户端的字节数，从而减少传输数据所需的时

间。您可以在 API 定义中启用内容编码，也可设置触发压缩的最小响应大小。默认情况下，API 不会启用内容编码支持。

设置比平均执行时间高出几秒的函数超时，以解决通信路径中使用的下游服务中的所有瞬时问题。这也同样适用于处理 Step Functions 活动、任务和 SQS 消息可见性的情况。

在 AWS Lambda 中选择默认内存设置和超时可能会对性能、成本和操作程序产生不良影响。

设置远高于平均执行时间的超时，可能会导致函数在代码出现故障时执行更长时间，从而导致更高的成本并可能达到并发限制，具体取决于调用此类函数的方式。

如果发生瞬时网络问题或下游服务异常，则设置等于函数成功执行一次的超时可能会触发无服务器应用程序突然停止执行。

在未执行负载测试的情况下设置超时，更重要的是在未考虑上游服务的情况下，只要任何部件首先达到超时就可能会导致错误。

遵循使用 Lambda 函数的最佳实践[54]，例如容器重用、根据运行需要最大限度地减小部署程序包大小，以及最大限度地降低依赖项（包括可能无法针对快速启动进行优化的框架）的复杂性。应始终考虑第 99 百分位 (P99) 延迟，因为 1 不会影响与其他团队商定的应用程序 SLA。

对于 VPC 中的 Lambda 函数，请避免对 VPC 中基础资源的公共主机名进行 DNS 解析。例如，如果您的 Lambda 函数访问 VPC 中的 Amazon RDS 数据库实例，则通过不可公开访问的选项启动该实例。

执行 Lambda 函数之后，AWS Lambda 会在任意时间内保持此次执行的上下文，以期再次进行 Lambda 函数调用。这样一来，您可以在全局范围内一次性进行成本高昂的操作，例如建立数据库连接或任何初始化逻辑。在后续调用中，您可以验证其是否仍然有效并重用现有连接。

异步事务

由于客户希望使用更加现代化、更具交互性的用户界面，因此您无法再使用同步事务来维系复杂的工作流。您需要的服务交互越多，最终进行链式调用的可能性就越大，这最终可能会增加服务稳定性和响应时间面临的风险。

现代化的 UI 框架（例如 Angular.js、VueJS 和 React）、异步事务以及云原生工作流程提供了一种可持续的方法，能够满足客户需求，同时有助于您将组件分离出来并专注于流程和业务领域。

这些异步事务（或者常被描述为事件驱动的架构）在云中启动下游的后续编排事件，而非将客户端约束在锁定等待状态（I/O 阻塞）以等待响应。异步工作流程可以处理多种使用案例，包括但不限于：数据提取、ETL 操作和订单/请求执行。

在这些使用案例中，系统会在数据到达时对其进行处理，并在数据变更时对其进行检索。我们概述了两种常见异步工作流程的最佳实践，您可以从中了解集成和异步处理的一些优化模式。

无服务器数据处理

在无服务器数据处理工作流程中，数据从客户端提取到 Kinesis（使用 Kinesis 代理、开发工具包或 API）并到达 Amazon S3。

新对象启动自动执行的 Lambda 函数。此函数常用于对数据进行转换或分区以做进一步处理，并且可能存储在其他目标（例如 DynamoDB 或数据处于其最终格式的另一个 S3 存储桶）中。

由于您可能会对不同数据类型进行不同的转换，因此我们建议您将转换细分为不同的 Lambda 函数，以实现最佳性能。通过这种方法，您可以灵活地并行运行数据转换，从而提高速度并降低成本。

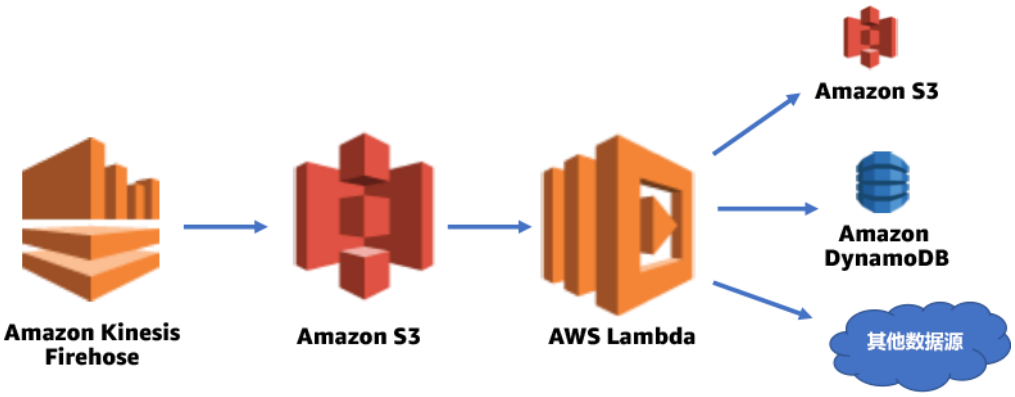

图 24：异步数据提取

Kinesis Data Firehose 提供了可用于替代 Lambda 的本机数据转换，无需额外的逻辑即可将 Apache 日志/系统日志中的记录转换为 CSV、JSON，并可将 JSON 转换为 Parquet 或 ORC。

无服务器事件提交与状态更新

假设您有一个电子商务网站，客户提交了一个启动库存扣减和发货流程的订单；或者企业应用程序提交了可能需要几分钟才能响应的大型查询。

完成此类常见事务所需的流程可能需要进行多次服务调用，这些调用可能需要几分钟才能完成。在这些调用中，您需要通过添加重试和指数回退来防范潜在的故障，但是，对于等待事务完成的人来说，这可能会导致欠佳的最佳用户体验。

对于与此类似的冗长而复杂的工作流程，您可以将 API Gateway 或 AWS AppSync 与 Step Functions 集成，后者一收到新的授权请求就会立即启动这一业务流程。Step Functions 能够使用执行 ID 对调用方（移动应用程序、开发工具包、Web 服务等）立即进行响应。

对于旧版系统，您可以使用执行 ID 通过其他 REST API 对 Step Functions 进行轮询，以获知业务流程状态。借助 WebSockets，无论您使用 REST 还是 GraphQL，都可以通过在工作流程的每个步骤中提供更新来实时接收业务流程状态。

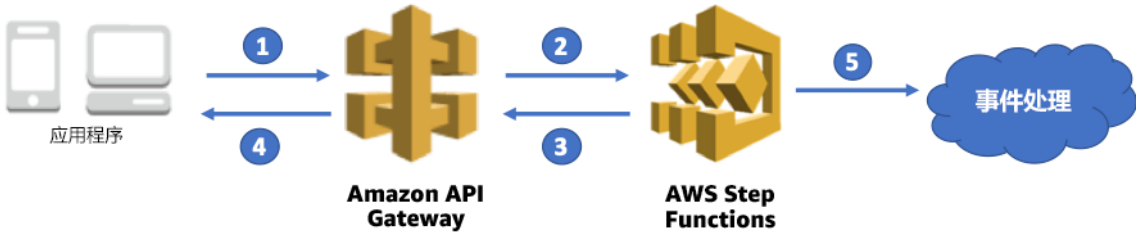

图 25：使用 Step Functions 状态机的异步工作流程

还有一种常见场景是直接将 API Gateway 与 SQS 或 Kinesis 集成，以作为扩展层。仅当期望调用者提供其他业务信息或自定义请求 ID 格式时，才需要使用 Lambda 函数。

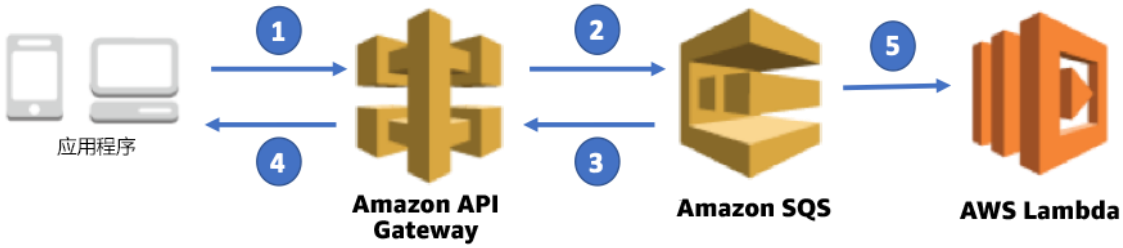

图 26：使用队列作为扩展层的异步工作流程

在第二个示例中，SQS 有多种用途：

1. 持久存储请求记录很重要，因为客户端知道请求最终将会得到处理，从而可以继续“安心地”执行整个工作流程

2. 发生可能会使后端暂时不堪负荷的突发事件时，可在资源变为可用的情况下对请求进行轮询以便处理请求。

与不存在队列的第一个示例相比，Step Functions 可以持久地存储数据，而无需借助队列或跟踪数据源状态。在这两个示例中，最佳实践都是在客户端提交请求之后执行异步工作流程，如果可能需要几分钟时间才能完成，还要避免将产生的响应用作阻塞代码。

借助 WebSockets，AWS AppSync 能够通过 GraphQL 订阅以开箱即用的方式提供这一功能。借助订阅，授权客户可以侦听其感兴趣的数据突变。这非常适合进行流式传输或可能产生多个响应的数据。

借助 AWS AppSync，随着 DynamoDB 中状态更新的变更，客户端可在发生更新时自动订阅和接收更新，这是数据驱动用户界面时的理想模式。

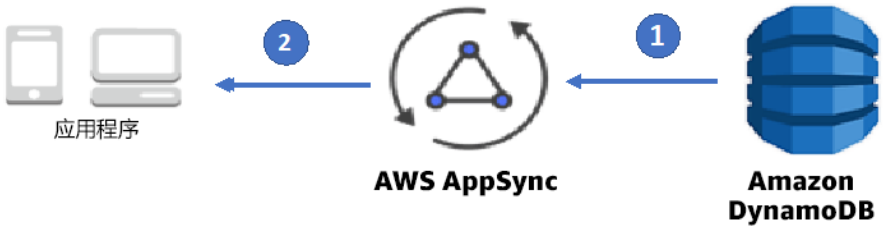

图 27：使用 AWS AppSync 和 GraphQL 通过 WebSockets 进行异步更新

Webhook 可通过 SNS 主题 HTTP 订阅实现。使用者可以托管一个 HTTP 终端节点，SNS 将在发生事件（例如数据文件到达 Amazon S3）后通过 POST 方法回调该终端节点。当客户端为可配置的客户端（例如可托管终端节点的其他微服务）时，适用该模式。另外，Step Functions 支持回调，状态机将阻止该回调，直到收到给定任务的响应。

图 28：使用 SNS 通过 Webhook 进行异步通知

最后，由于多个客户端会持续轮询 API 以获取状态，因此从成本和资源角度考虑，采用轮询可能会产生较高的成本。如果因为环境限制而只能使用轮询，则最佳实践是与客户端建立 SLA 以限制“空轮询”的数量。

图 29：客户端轮询最近的事务中发生的更新

例如，如果大型数据仓库查询平均需要两分钟的响应时间，当数据不可用时，客户端应在两分钟后使用指数回退方法对 API 进行轮询。通过两种常见模式可以确保客户端进行轮询的频率不会超出预期：设置限流，以及设置可以再次进行安全轮询的时间的时间戳。

对于时间戳方式，被轮询的系统可以返回带有时间戳或时间段的额外字段，用于指示使用者可以再次进行安全轮询的时间。此方法需要使用者合理使用；如果出现滥用，您还可以通过设置限流来实现更完整的实施。

审核

有关适用于无服务器应用程序的性能效率提升方法中**审核**领域的最佳实践，请参阅“AWS 架构完善的框架”白皮书。

监控

有关适用于无服务器应用程序的性能效率提升方法中**监控**领域的最佳实践，请参阅“AWS 架构完善的框架”白皮书。

权衡

有关适用于无服务器应用程序的性能效率提升方法中**权衡**领域的最佳实践，请参阅“AWS 架构完善的框架”白皮书。

关键 AWS 服务

提升性能效率所需的关键 AWS 服务包括 DynamoDB Accelerator、API Gateway、Step Functions、NAT 网关、Amazon VPC 和 Lambda。

资源

请参阅以下资源，详细了解提升性能效率的最佳实践。

文档和博客

- AWS Lambda 常见问题[55]
- 使用 AWS Lambda 函数的最佳实践[56]
- AWS Lambda 工作原理[57]
- Understanding Container Reuse in AWS Lambda[58]
- 配置 Lambda 函数以访问 Amazon VPC 中的资源[59]
- 启用 API 缓存以增强响应能力[60]
- DynamoDB：全局二级索引[61]
- Amazon DynamoDB Accelerator (DAX)[62]
- 开发人员指南：Kinesis Streams[63]
- Java 开发工具包：性能提升配置
- Node.js 开发工具包：启用 HTTP 保持活动状态
- Node.js 开发工具包：改进导入
- 使用 Amazon SQS 队列和 AWS Lambda 获得高吞吐量

- Increasing stream processing performance with enhanced fan-out
- Lambda Power Tuning
- Amazon DynamoDB 按需模式和预置模式的适用场景
- 使用 Amazon CloudWatch Logs Insights 分析日志数据
- 将多种数据源与 AWS AppSync 集成
- Step Functions 服务集成
- 缓存模式
- 缓存无服务器应用程序
- 将 Amazon Athena 与 AWS Glue 结合使用时的最佳实践

成本优化支柱

成本优化支柱包括系统在整个生命周期中不断完善和改进的过程。从最开始的概念验证的初始设计到生产工作负载的持续运营，您都可以采用本文档中的实践来构建和运营具有成本意识的系统，从而在实现业务成果的同时最小化成本，使您的企业能够最大限度地提高投资回报率。

定义

云中的成本优化包括四个领域的最佳实践：

- 资源成本效益
- 供需匹配
- 支出认知
- 持续优化

与其他支柱一样，成本优化支柱也需要权衡各种因素。例如，您想优化上市速度还是优化成本？在某些情况下，最好优化上市速度以便快速上市、交付新功能或只是为了按时完成任务，而不是优化预付成本。

由于人们总是倾向于以过度补偿的方式来“以防万一”，而不是花时间进行基准测试，逐渐找出成本最优的部署，导致设计决策有时会过于仓促，缺乏对经验数据的参考。

这通常会导致明显的过度预置和优化不足的部署。以下各节介绍了一些技巧和战略性指导，以帮助您实现初始部署和持续成本优化。

通常，使用无服务器架构有助于降低成本，因为某些服务（例如 AWS Lambda）在闲置时不会产生任何成本。但是，通过遵循某些最佳实践并进行权衡将有助于您进一步降低这些解决方案的成本。

最佳实践

成本 1：如何优化成本？

资源成本效益

从合理分配资源的角度考虑，无服务器架构更易于管理。无服务器架构采用按价值付费的定价模式并支持按需调整规模，因而能够有效减少容量规划工作。

如卓越运营和性能支柱章节中所述，优化无服务器应用程序会直接影响其产生的价值及所需成本。

Lambda 根据内存情况按比例分配 CPU、网络和存储 IOPS，同时采用以每 100 毫秒为单位的计费增量维度，因此执行速度越快，成本越低，您的函数也越能产生更大的价值。

供需匹配

AWS 无服务器架构旨在按需进行扩展，因此没有可遵循的适用实践。

支出认知

如 AWS 架构完善的框架中所述，通过云，您可以获得更大的灵活性和敏捷性，从而支持创新以及快速的开发和部署。这样便节省了自建本地基础设施所需的人工环节和时间，包括确定硬件规格、协商报价、管理购买订单、安排发货和部署资源。

随着无服务器架构的扩展，Lambda 函数、API、阶段以及其他资产的数量将成倍增加。这些架构大多需要对成本和资源管理进行预算和预测，而使用标签可助您一臂之力。您可以将 AWS 账单中的成本分配给各个函数和 API，并在 AWS Cost Explorer 中获得各项目所用成本的细化视图。

一种有效的实施方式是以编程方式对属于同一项目的资产共享相同的键值标签，并根据您创建的标签生成自定义报告。此功能不仅可以帮助您分配成本，还可以确定哪些资源属于哪些项目。

持续优化

有关适用于无服务器应用程序的成本优化方法中**持续优化**领域的最佳实践，请参阅“AWS 架构完善的框架”白皮书。

日志记录提取和存储

AWS Lambda 使用 CloudWatch Logs 存储执行的输出，以识别执行中的问题并进行故障排除，同时监控无服务器应用程序。这将在提取和存储两方面影响 CloudWatch Logs 服务的成本。

设置适当的日志记录级别并删除不必要的日志记录信息以优化日志提取。在调试模式下，使用环境变量来控制应用程序日志记录级别和示例日志记录，确保您在必要时可以获取更多见解。

为新的和现有的 CloudWatch Logs 组设置日志保留期限。对于日志存档，请导出并设置最适合您需求的经济高效的存储类别。

直接集成

如果您的 Lambda 函数在与其他 AWS 服务集成时未执行自定义逻辑，则可能说明没有必要使用自定义逻辑。

API Gateway、AWS AppSync、Step Functions、EventBridge 和 Lambda Destinations 可直接与多种服务集成，在降低运营开销的同时为您提供更多价值。

如微服务场景中所述，大多数公共无服务器应用程序都为 API 提供了与所提供的协定无关的实施。

一个比较适合直接集成的示例场景是通过 REST API 提取点击流数据。

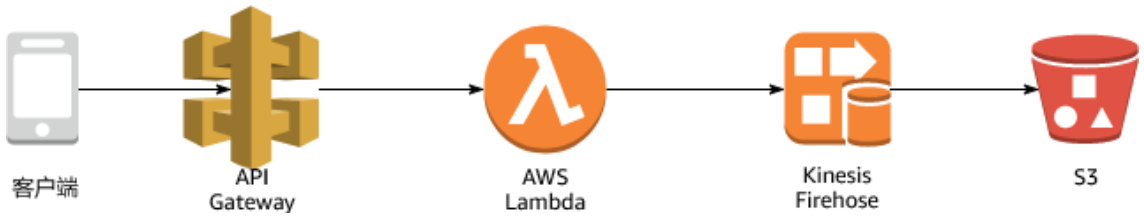

图 30：使用 Kinesis Data Firehose 将数据发送到 Amazon S3

在此场景中，API Gateway 将执行 Lambda 函数，该函数可轻松地将传入记录提取到 Kinesis Data Firehose 中，在其中将对记录进行批处理，随后再将记录存储到 S3 存储桶中。由于此示例不需要其他逻辑，因此我们可以使用 API Gateway 服务代理直接与 Kinesis Data Firehose 进行集成。

图 31：通过实施 AWS 服务代理来降低向 Amazon S3 发送数据的成本

通过这种方法，我们在 API Gateway 中实施 AWS 服务代理，从而消除了因使用 Lambda 和不必要的调用而产生的成本。同时，如果需要多个分区来满足提取速率，则此方法可能会带来一些额外的复杂性。

如果对延迟有较高要求，您可以通过提供适用的凭证，将数据直接流式传输到 Kinesis Data Firehose，但这样做会对抽象、协定和 API 功能造成影响。

图 32：通过直接使用 Kinesis Data Firehose 开发工具包进行流式传输来降低向 Amazon S3 发送数据的成本

对于需要与您的 VPC 中或本地的内部资源连接并且不需要自定义逻辑的场景，请使用 API Gateway 私有集成。

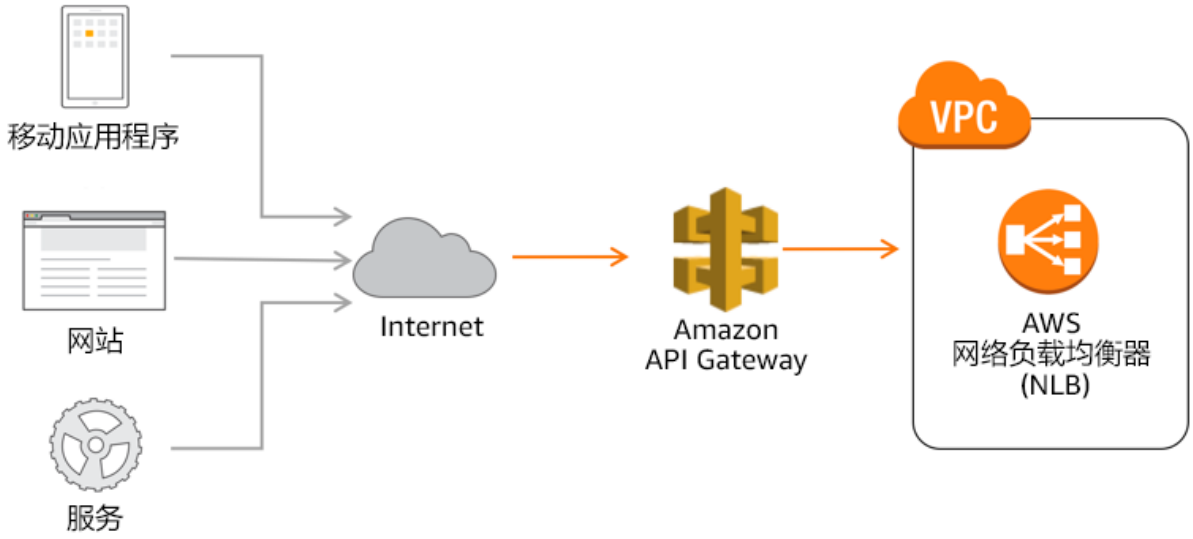

图 33：在 VPC 中使用 Amazon API Gateway 私有集成而非 Lambda 访问私有资源

通过这种方法，API Gateway 可将每个传入请求发送到您在 VPC 中拥有的内部网络负载均衡器，该均衡器可以通过 IP 地址将流量转发到同一 VPC 中或本地的任何后端。

无需额外的跃点就可以将请求发送到私有后端，使得这种方法兼具成本与性能优势，同时还具有授权、限流和缓存机制等附加优势。

另一种场景是扇出模式，Amazon SNS 使用这种模式向其所有订阅者发送消息广播。这种方法需要其他应用程序逻辑来筛选并避免不必要的 Lambda 调用。

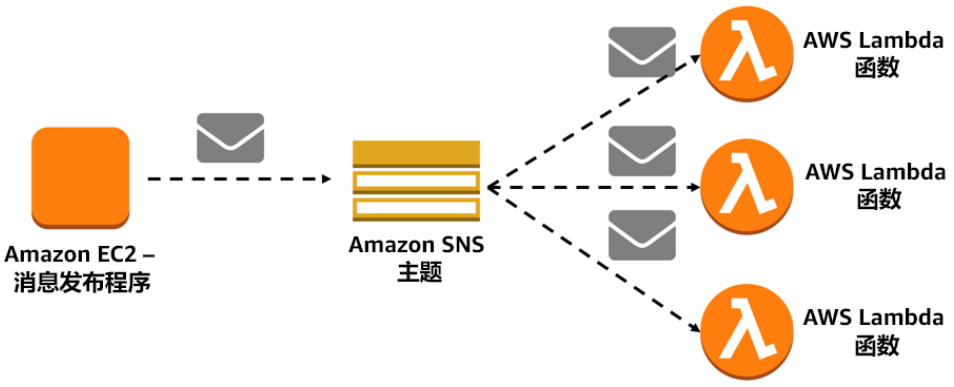

图 34：未经消息属性筛选的 Amazon SNS

SNS 可以根据消息属性对事件进行筛选，并更高效地将消息传递给正确的订阅者。

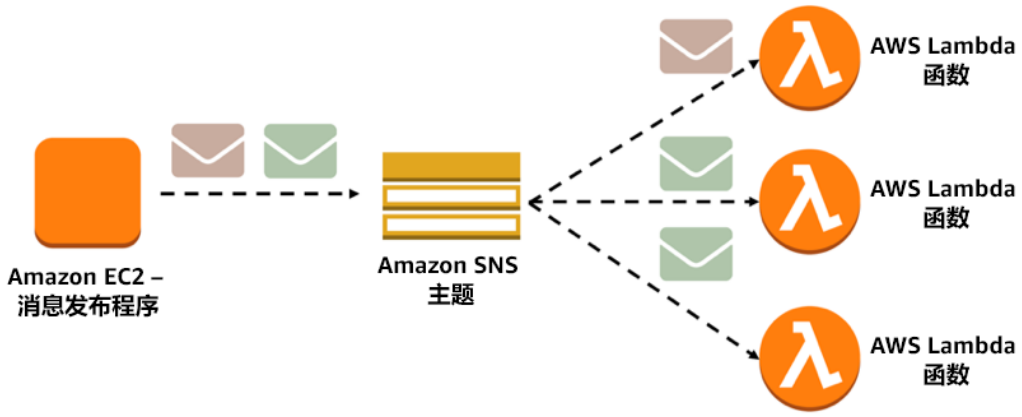

图 35：经过消息属性筛选的 Amazon SNS

另一个示例是需要长时间运行的处理任务，您可能需要等待这些任务完成才能执行后续步骤。此等待状态可以在 Lambda 代码中实施，但是，无论是使用事件转换为异步处理，还是使用 Step Functions 实施等待状态都要更加高效。

例如，在下图中，我们对 AWS Batch 作业进行轮询，并每 30 秒检查一次状态，看看它是否已完成。我们未在 Lambda 函数中编码此等待，而是实施轮询 (*GetJobStatus*) + 等待 (*Wait30Seconds*) + 决策程序 (*CheckJobStatus*)。

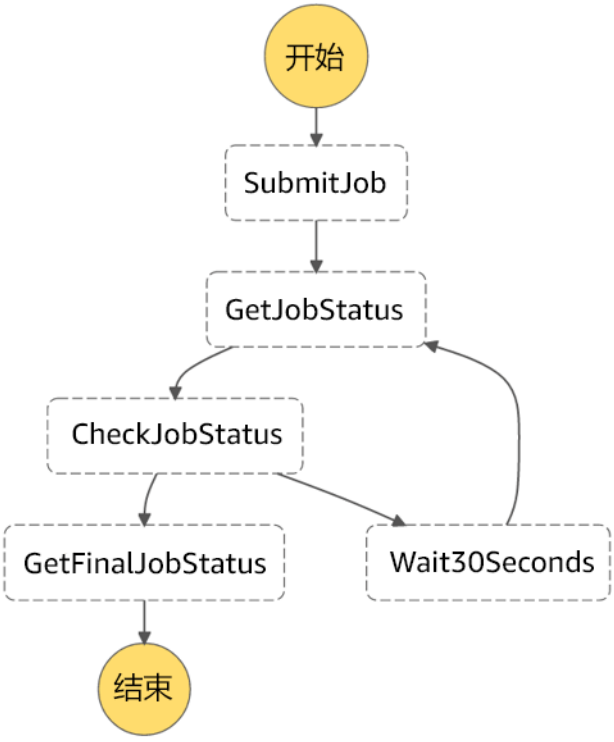

图 36：使用 AWS Step Functions 实施等待状态

使用 Step Functions 实施等待状态不会产生任何其他成本，因为 Step Functions 的定价模式基于状态间转换，而不是基于某一状态内花费的时间。

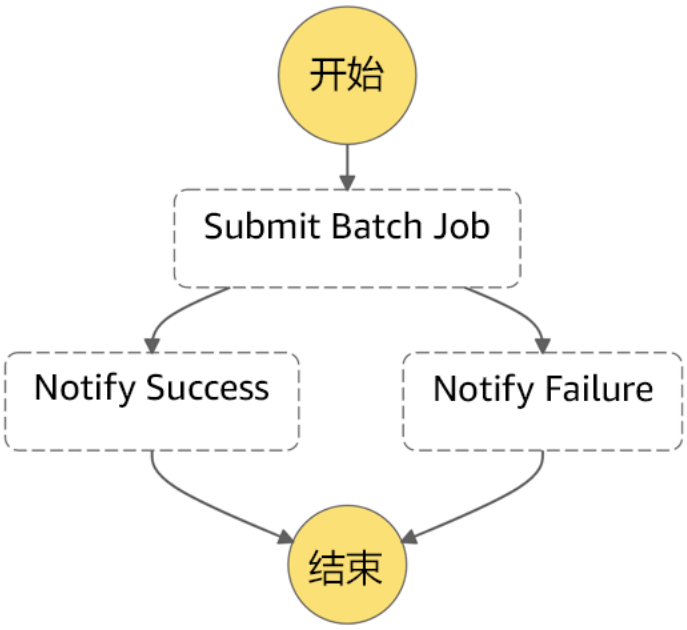

图 37：Step Functions 服务集成同步等待

根据必须等待的集成的情况，Step Functions 可以同步等待，然后再继续执行下一个任务，从而节省了额外转换时间。

代码优化

如性能支柱章节中所述，优化无服务器应用程序可以有效提高每次执行产生的价值。

通过使用全局变量来维护与您的数据存储或其他服务和资源的连接，可以提高性能并减少执行时间，同时降低成本。有关更多信息，请参阅性能支柱章节。

使用托管服务功能可以提高每次执行产生的价值，从 Amazon S3 检索和筛选对象便是此类示例之一，因为从 Amazon S3 提取大型对象需要更大的内存来处理 Lambda 函数。

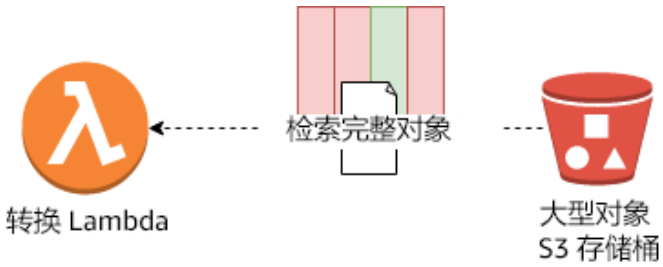

图 38：检索完整 S3 对象的 Lambda 函数

在上图中，我们可以看到，从 Amazon S3 检索大型对象时，我们可能会增加 Lambda 的内存消耗并增加执行时间（用于进行函数转换、迭代或收集所需的数据），在某些情况下，仅需要其中的部分信息。

这在图中分别用红色的三列（不需要数据）和绿色的一列（需要数据）进行表示。使用 Athena SQL 查询来收集执行所需的详细信息，可以减少执行转换所需的检索时间和对象大小。

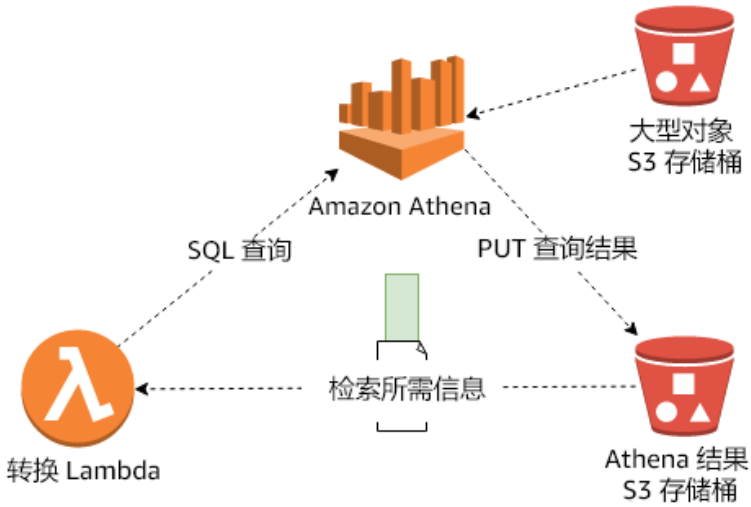

图 39：Lambda 与 Athena 对象检索

在下图中，我们可以看到，通过查询 Athena 来获取特定数据，我们可以减小检索到的对象的大小，这样做还有额外的好处：因为 Athena 将其查询结果保存在 S3 存储桶中，并在结果异步到达 Amazon S3 时调用 Lambda 调用，使得我们可以重用该内容。

使用 S3 Select 也可实现类似的方法。使用 S3 Select，应用程序可以通过使用简单的 SQL 表达式仅从对象中检索数据子集。与前面使用 Athena 的示例一样，从 Amazon S3 检索较小的对象可以减少执行时间，同时减少 Lambda 函数的内存使用量。

200 秒

```
# Download and process all keys
for key in src_keys:
  response =
s3_client.get_object(Bucket=src_bucket,
Key=key)
  contents = response['Body'].read()
  for line in contents.split('\n')[:-1]:
      line_count +=1
      try:
          data = line.split(',')
           srcIp = data[0][:8]


….
```

95 秒

```
# Select IP Address and Keys
for key in src_keys:
  response =
s3_client.select_object_content
      (Bucket=src_bucket, Key=key,
expression =
       SELECT SUBSTR(obj._1, 1, 8),
obj._2 FROM s3object as obj)
  contents = response['Body'].read()
  for line in contents:
      line_count +=1
      try:


….
```

图 40：使用 Amazon S3 与使用 S3 Select 的 Lambda 性能统计对比

资源

请参阅以下资源，详细了解成本优化的最佳实践。

文档和博客

- CloudWatch Logs 保留[64]
- 将 CloudWatch Logs 导出到 Amazon S3[65]
- 将 CloudWatch Logs 流式传输到 Amazon ES[66]
- 在 Step Functions 状态机中定义等待状态[67]
- Coca-Cola Vending Pass State Machine Powered by Step Functions[68]
- Building high throughput genomics batch workflows on AWS[69]
- Simplify your Pub/Sub Messaging with Amazon SNS Message Filtering
- S3 Select 与 Glacier Select
- 适用于 MapReduce 的 Lambda 参考架构
- 无服务器应用程序存储库应用程序 – 自动设置 CloudWatch Logs 组保留
- Ten resources every Serverless Architect should know

白皮书

- 使用无服务器架构优化企业经济[70]

总结

尽管无服务器应用程序能够使开发人员摆脱千篇一律的繁重工作，但仍然需要注意一些重要原则。

在可靠性方面，定期对故障路径进行测试有助于您在投入生产之前及早发现错误。在性能方面，从客户期望出发将有助于设计出最佳体验。同时还有许多 AWS 工具可帮助优化性能。

在成本优化方面，您可以根据流量需求调整资源大小，从而减少无服务器应用程序中不必要的浪费，并通过优化应用程序来提高产出价值。在运营方面，您的架构应尽可能实现事件的自动化响应。

最后，安全的应用程序能够保护您组织的敏感信息资产，同时能够满足各层的任何合规性要求。

随着工具和流程的生态系统日趋成熟，无服务器应用程序的格局正在不断发展壮大。届时，我们会继续更新本文档，帮助您确保无服务器应用程序保持完善的架构。

贡献者

以下是对本文档做出贡献的个人和组织：

- Adam Westrich：Amazon Web Services 高级解决方案架构师
- Mark Bunch：Amazon Web Services 企业解决方案架构师
- Ignacio Garcia Alonso：Amazon Web Services 解决方案架构师
- Heitor Lessa：Amazon Web Services Well-Architected 首席无服务器项目主管
- Philip Fitzsimons：Amazon Web Services Well-Architected 高级经理
- Dave Walker：Amazon Web Services 首席专家解决方案架构师
- Richard Threlkeld：Amazon Web Services 移动部门高级产品经理
- Julian Hambleton-Jones：Amazon Web Services 高级解决方案架构师

延伸阅读

有关更多信息，请参阅以下内容：

- AWS 架构完善的框架[71]

文档修订

日期	描述
2019 年 12 月	全面更新：添加了新功能并完善了最佳实践。
2018 年 11 月	添加了适用于 Alexa 和移动设备的新场景。进行全面更新，以反映新功能并完善最佳实践。
2017 年 11 月	原始版本。

备注

[1] https://aws.amazon.com/well-architected

[2] http://d0.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

[3] https://github.com/alexcasalboni/aws-lambda-power-tuning

[4] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

[5] http://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/es-managedomains.html

[6] https://www.elastic.co/guide/en/elasticsearch/guide/current/scale.html

[7] http://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-scaling.html

[8] https://d0.awsstatic.com/whitepapers/whitepaper-streaming-data-solutions-on-aws-with-amazon-kinesis.pdf

[9] http://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecords.html

[10] http://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-duplicates.html

[11] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html#stream-events

[12] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html

[13] http://docs.aws.amazon.com/apigateway/latest/developerguide/stage-variables.html

[14] http://docs.aws.amazon.com/lambda/latest/dg/env_variables.html

[15] https://github.com/awslabs/serverless-application-model

[16] https://aws.amazon.com/blogs/aws/latency-distribution-graph-in-aws-x-ray/

[17] http://docs.aws.amazon.com/lambda/latest/dg/lambda-x-ray.html

[18] http://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html

[19] https://aws.amazon.com/blogs/compute/continuous-deployment-for-serverless-applications/

[20] https://github.com/awslabs/aws-serverless-samfarm

[21] https://d0.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf

[22] https://aws.amazon.com/serverless/developer-tools/

[23] http://docs.aws.amazon.com/lambda/latest/dg/with-s3-example-create-iam-role.html

[24] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-method-request-validation.html

[25] http://docs.aws.amazon.com/apigateway/latest/developerguide/use-custom-authorizer.html

[26] https://aws.amazon.com/blogs/compute/secure-api-access-with-amazon-cognito-federated-identities-amazon-cognito-user-pools-and-amazon-api-gateway/

[27] http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[28] https://aws.amazon.com/pt/articles/using-squid-proxy-instances-for-web-service-access-in-amazon-vpc-another-example-with-aws-codedeploy-and-amazon-cloudwatch/

[29] https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf

[30] https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Best_Practices.pdf

[31] https://www.twistlock.com/products/serverless-security/

[32] https://snyk.io/

[33] https://www.owasp.org/index.php/OWASP_Dependency_Check

[34] http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[35] http://docs.aws.amazon.com/lambda/latest/dg/limits.html

[36] http://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html#api-gateway-limits

[37] http://docs.aws.amazon.com/streams/latest/dev/service-sizes-and-limits.html

[38] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

[39] http://docs.aws.amazon.com/step-functions/latest/dg/limits.html

[40] https://aws.amazon.com/blogs/compute/error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[41] https://aws.amazon.com/blogs/compute/serverless-testing-with-aws-lambda/

[42] http://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-logs.html

[43] http://docs.aws.amazon.com/lambda/latest/dg/versioning-aliases.html

[44] http://docs.aws.amazon.com/apigateway/latest/developerguide/stages.html

[45] http://docs.aws.amazon.com/general/latest/gr/api-retries.html

[46] http://docs.aws.amazon.com/step-functions/latest/dg/tutorial-handling-error-conditions.html#using-state-machine-error-conditions-step-4

[47] http://docs.aws.amazon.com/xray/latest/devguide/xray-services-lambda.html

[48] http://docs.aws.amazon.com/lambda/latest/dg/dlq.html

[49] https://aws.amazon.com/blogs/compute/error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[50] http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[51] http://microservices.io/patterns/data/saga.html

[52] http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[53] https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf

[54] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[55] https://aws.amazon.com/lambda/faqs/

[56] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[57] http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html

[58] https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[59] http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[60] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html

[61] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html

[62] https://aws.amazon.com/dynamodb/dax/

[63] http://docs.aws.amazon.com/streams/latest/dev/amazon-kinesis-streams.html

[64] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/SettingLogRetention.html

[65] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/S3ExportTasksConsole.html

[66] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_ES_Stream.html

[67] http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[68] https://aws.amazon.com/blogs/aws/things-go-better-with-step-functions/

[69] https://aws.amazon.com/blogs/compute/building-high-throughput-genomics-batch-workflows-on-aws-workflow-layer-part-4-of-4/

[70] https://d0.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf

[71] https://aws.amazon.com/well-architected