aws

云设计模式、架构和实施

AWS 规范性指导

AWS 规范性指导: 云设计模式、架构和实施

Table of Contents

云设计模式、架构和实施

Anitha Deenadayalan，亚马逊云科技（AWS）

2024 年 5 月（文档历史记录）

本指南为使用 AWS 服务实现常用的现代化设计模式提供了指引。越来越多的现代应用程序是通过使用微服务架构进行设计的，从而实现可扩展性、提升发布速度、缩小更改的影响范围和减少回归。这可以提高开发人员的工作效率，提高敏捷性，使之更好地进行创新，并更加专注于业务需求。微服务架构还支持对服务和数据库使用最佳技术，且促进多语言代码和多语言持久性。

传统上，单体应用程序在单个进程中运行，使用一个数据存储，并在垂直扩展的服务器上运行。相比之下，现代微服务应用程序是细粒度的，具有独立的故障域，作为服务在网络上运行，并且可以根据应用场景使用多个数据存储。这些服务可以水平扩展，单个事务可能跨越多个数据库。使用微服务架构开发应用程序时，开发团队必须专注于网络通信、多语言持久性、水平扩展、最终一致性，以及跨数据存储的事务处理。因此，现代化模式对于解决现代应用程序开发中的常见问题至关重要，这些模式有助于加快软件交付。

本指南为云架构师、技术主管、应用程序和企业所有者以及希望根据架构完善的最佳实践为设计模式选择合适云架构的开发人员提供了技术参考。本指南中讨论的每种模式都应对微服务架构中的一个或多个已知场景。该指南讨论了与每种模式相关的问题和注意事项，提供了高级架构实现，并描述了各模式的亚马逊云科技实施。在存在资源时，还会提供开源 GitHub 示例和研讨会链接。

该指南涵盖了以下模式：

- 防腐层
- API 路由模式：
  - 主机名路由
  - 路径路由
  - HTTP 标头路由
- 断路器
- 事件溯源
- 六边形架构
- 发布/订阅
- 使用退避重试
- saga 模式：

  - saga 编配
  - saga 编排
- 分散-收集
- Strangler fig
- 事务发件箱

目标业务成果

通过使用本指南中所讨论的模式，对应用程序进行现代化改造，您可以：

- 设计和实施针对成本和性能优化后的可靠、安全、操作高效的架构。
- 缩短需要这些模式的应用场景的周期时间，以便您转而专注于组织特定的挑战。
- 使用亚马逊云科技服务对模式实施进行标准化，从而加快开发速度。
- 帮助您的开发人员在无需继承技术债务的情况下构建现代应用程序。

防腐层模式

意图

防腐层（ACL）模式充当中介层，将域模型语义从一个系统转换为另一个系统。在使用上游团队建立的通信合同之前，它将上游限界上下文（单体）的模型转换为适合下游限界上下文（微服务）的模型。当下游限界上下文包含核心子域，或者上游模型是不可修改的遗留系统时，这种模式可能适用。它还能降低转型风险和业务中断，方法是防止在调用方必须透明地将调用重定向到目标系统时对其进行更改。

动机

在迁移过程中，当单体应用程序迁移到微服务时，新迁移的服务的域模型语义可能会发生更改。当需要使用单体架构中的功能来调用这些微服务时，应将调用路由到迁移的服务，而无需对调用服务进行任何更改。ACL 模式允许单体通过充当适配器或将调用转换为更新的语义的 Facade 层来透明地调用微服务。

适用性

在以下情况下，考虑使用此模式：

- 您现有的单体应用程序必须与已迁移到微服务的功能进行通信，并且迁移的服务域模型和语义与原始功能不同。
- 两个系统具有不同的语义，且需要交换数据，但修改一个系统使其与另一个系统兼容是不切实际的。
- 您想使用一种快速简化的方法来使一个系统适应另一个系统，同时将影响降至最低。
- 您的应用程序正在与外部系统通信。

问题和注意事项

- 团队依赖关系：当系统中的不同服务由不同的团队拥有时，迁移服务中的新域模型语义可能会导致调用系统发生变化。但是，团队可能无法以协调的方式进行这些更改，因为他们可能还有其他优先事项。ACL 解耦被调用方并转换调用以匹配新服务的语义，从而无需调用方在当前系统中进行更改。
- 运营开销：ACL 模式需要额外的工作来操作和维护。此工作包括将 ACL 与监控和提醒工具、发布流程以及持续集成和持续交付（CI/CD）流程集成。

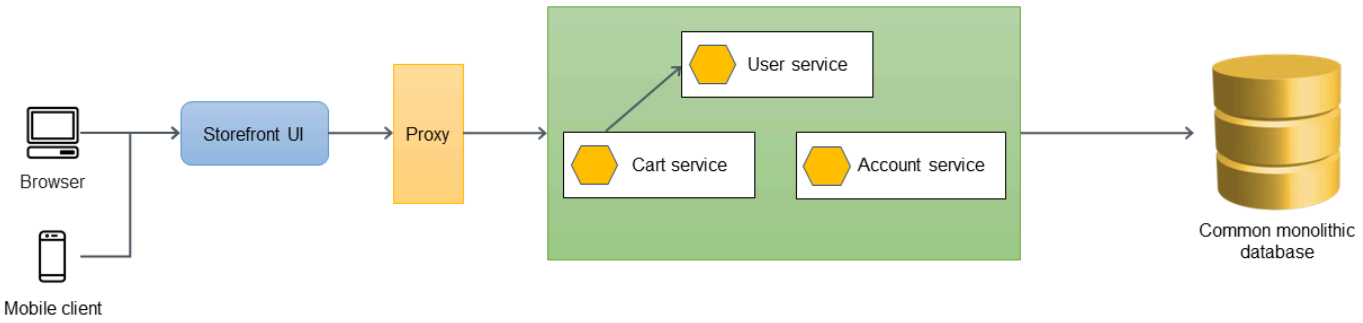
- 单点故障：ACL 中的任何故障都可能使目标服务无法访问，从而导致应用程序问题。为了缓解此问题，您应该内置重试功能和断路器。请参阅使用回退重试和断路器模式，以了解有关这些选项的更多信息。设置适当的提醒和日志记录将缩短平均解决时间（MTTR）。
- 技术债务：作为迁移或现代化战略的一部分，请考虑 ACL 是暂时或临时解决方案，还是长期解决方案。如果是临时解决方案，您应将 ACL 记录为技术债务，并在所有依赖调用方迁移完毕后将其停用。
- 延迟：由于请求需要从一个接口转换到另一个接口，额外层可能会带来延迟。我们建议您在将 ACL 部署到生产环境之前，先定义和测试对响应时间敏感的应用程序的性能容错能力。
- 扩展瓶颈：在服务可以扩展到峰值负载的高负载应用程序中，ACL 可能会成为瓶颈，并可能导致扩展问题。如果目标服务按需扩展，您应设计 ACL 以相应地进行扩展。
- 特定于服务或共享的实现：您可以将 ACL 设计为共享对象，以将调用转换和重定向到多个服务或特定于服务的类别。在确定 ACL 的实现类型时，请考虑延迟、扩展和容错能力。

实现

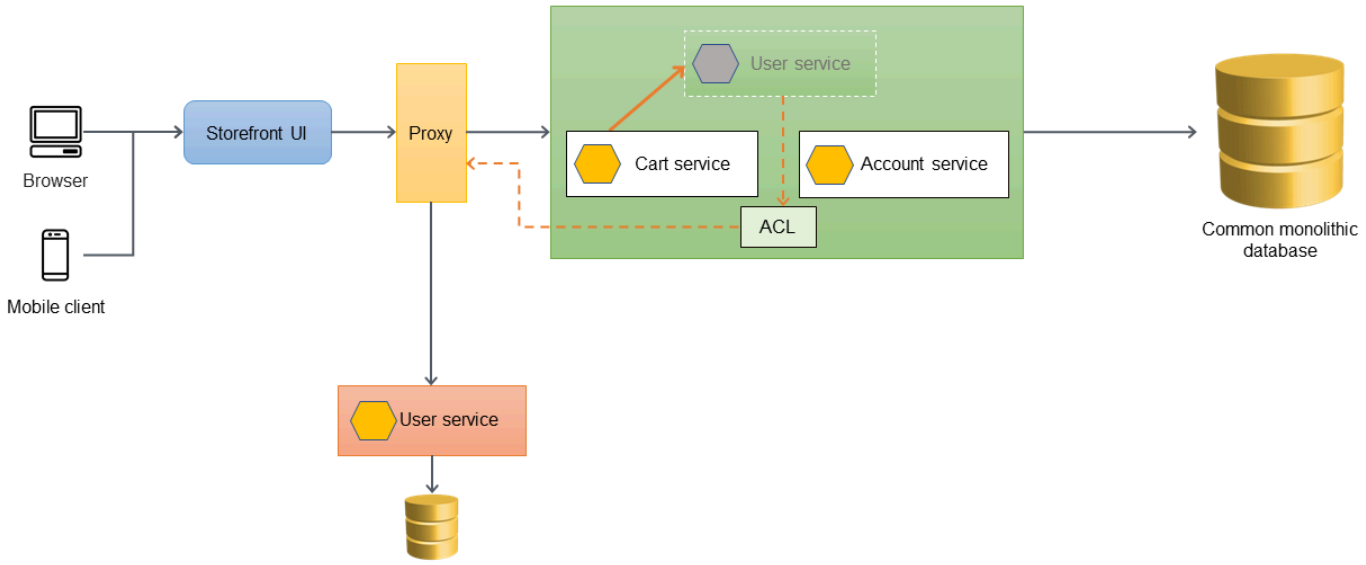
您可以在单体应用程序中将 ACL 作为特定于要迁移的服务的类别或独立的服务来实现。在所有从属服务都迁移到微服务架构后，必须停用 ACL。

高级架构

在以下示例架构中，单体应用程序具有三项服务：用户服务、购物车服务和账户服务。购物车服务依赖于用户服务，应用程序使用单体关系数据库。

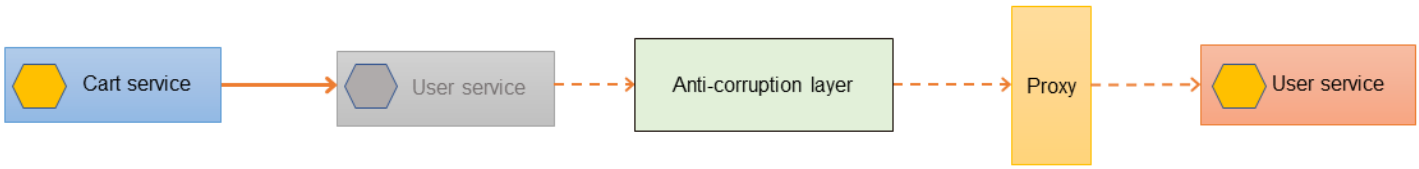


在以下架构中，用户服务已迁移到新的微服务。购物车服务调用用户服务，但在单体中不再提供该实现。 新迁移的服务的接口也可能与其之前（在单体应用程序中时）的接口不匹配。



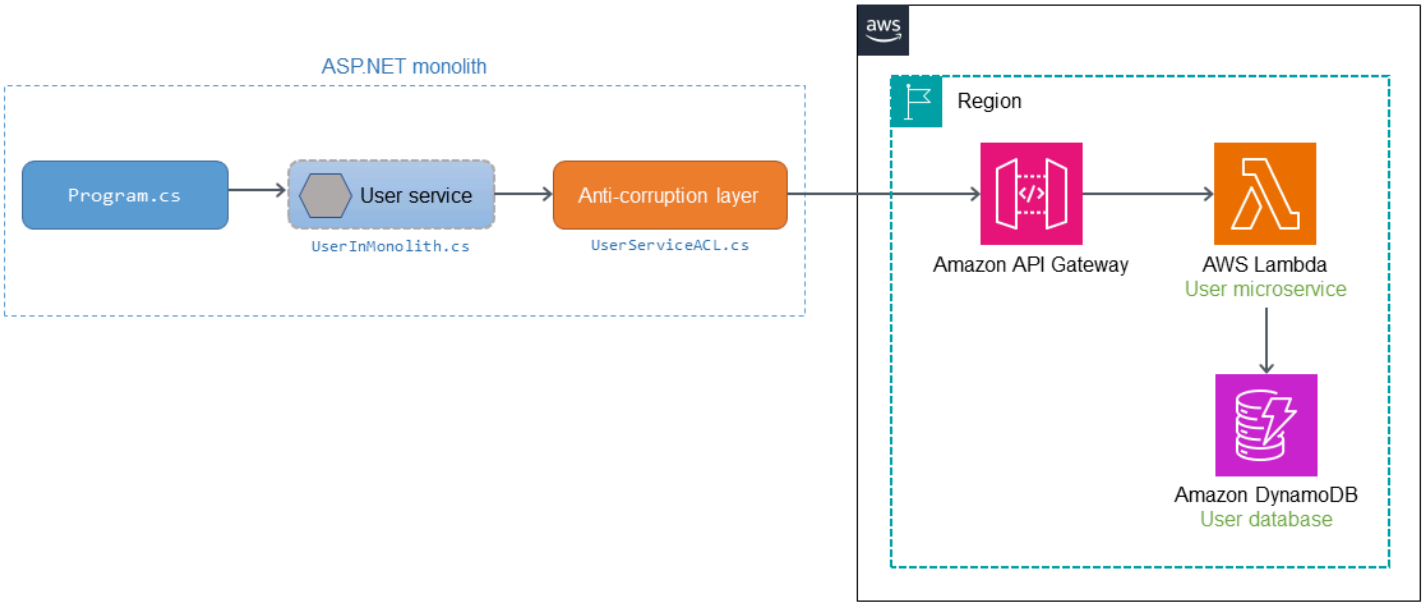
如果购物车服务必须直接调用新迁移的用户服务，则需要更改购物车服务并对单体应用程序进行全面测试。这可能会增加转型风险和业务中断。目标应该是最大限度地减少对单体应用程序现有功能的更改。

在这种情况下，我们建议您在旧的用户服务和新迁移的用户服务之间引入 ACL。ACL 可用作适配器或 Facade，以将调用转换为较新的接口。ACL 可以在单体应用程序内作为特定于已迁移服务的类别（例如 `UserServiceFacade` 或 `UserServiceAdapter`）来实现。在所有从属服务都迁移到微服务架构后，必须停用防腐层。



使用 AWS 服务来实施

下图显示了如何使用 AWS 服务实现此 ACL 示例。



用户微服务从 ASP.NET 单体应用程序中迁移出来，并作为 AWS Lambda 函数部署在 AWS 上。对 Lambda 函数的调用通过 Amazon API Gateway 进行路由。ACL 部署在单体中，用于转换调用以适应用户微服务的语义。

在 `Program.cs` 调用单体内的用户服务（`UserInMonolith.cs`）时，该调用会路由到 ACL（`UserServiceACL.cs`）。ACL 将调用转换为新的语义和接口，并通过 API Gateway 端点调用微服务。调用方（`Program.cs`）不知道用户服务和 ACL 中发生的转换和路由。由于调用方不知道代码更改，因此业务中断减少了，转型风险也降低了。

代码示例

以下代码片段提供了对原始服务的更改及 `UserServiceACL.cs` 的实现。收到请求后，原始用户服务会调用 ACL。ACL 会将源对象转换为与新迁移服务的接口匹配，调用该服务，并将响应返回给调用方。

```
public class UserInMonolith: IUserInMonolith
{
    private readonly IACL _userServiceACL;
    public UserInMonolith(IACL userServiceACL) => (_userServiceACL) = (userServiceACL);
    public async Task<HttpStatusCode> UpdateAddress(UserDetails userDetails)
    {
        //Wrap the original object in the derived class
        var destUserDetails = new UserDetailsWrapped("user", userDetails);
        //Logic for updating address has been moved to a microservice
        return await _userServiceACL.CallMicroservice(destUserDetails);
    }
```

```
}

public class UserServiceACL: IACL
{
    static HttpClient _client = new HttpClient();
    private static string _apiGatewayDev = string.Empty;

    public UserServiceACL()
    {
        IConfiguration config = new
 ConfigurationBuilder().AddJsonFile(AppContext.BaseDirectory + "../../../
config.json").Build();
        _apiGatewayDev = config["APIGatewayURL:Dev"];
        _client.DefaultRequestHeaders.Accept.Add(new
 MediaTypeWithQualityHeaderValue("application/json"));
    }
    public async Task<HttpStatusCode> CallMicroservice(ISourceObject details)
    {
        _apiGatewayDev +=  "/" + details.ServiceName;
        Console.WriteLine(_apiGatewayDev);

        var userDetails = details as UserDetails;
        var userMicroserviceModel = new UserMicroserviceModel();
        userMicroserviceModel.UserId = userDetails.UserId;
        userMicroserviceModel.Address = userDetails.AddressLine1 + ", " +
 userDetails.AddressLine2;
        userMicroserviceModel.City = userDetails.City;
        userMicroserviceModel.State = userDetails.State;
        userMicroserviceModel.Country = userDetails.Country;

        if (Int32.TryParse(userDetails.ZipCode, out int zipCode))
        {
            userMicroserviceModel.ZipCode = zipCode;
            Console.WriteLine("Updated zip code");
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
            return HttpStatusCode.BadRequest;
        }

        var jsonString =
 JsonSerializer.Serialize<UserMicroserviceModel>(userMicroserviceModel);
        var payload = JsonSerializer.Serialize(userMicroserviceModel);
```

```
        var content = new StringContent(payload, Encoding.UTF8, "application/json");

        var response = await _client.PostAsync(_apiGatewayDev, content);
        return response.StatusCode;
    }
}
```

GitHub 存储库

有关此模式示例架构的完整实施，请参阅 GitHub 存储库 https://github.com/aws-samples/anti-corruption-layer-pattern。

相关内容

- strangler fig 模式
- 断路器模式
- 使用回退重试模式

API 路由模式

在敏捷开发环境中，自治团队（例如小队和部落）拥有一项或多项包含许多微服务的服务。这些团队将这些服务作为 API 公开，以允许其使用者与其服务和操作组进行交互。

使用主机名和路径向上游使用者公开 HTTP API 有三种主要方法：

方法	描述	示例：
主机名路由	以主机名公开每一项服务。	`billing.api.example.com`
路径路由	以路径公开每一项服务。	`api.example.com/billing`
基于标头的路由	将每项服务作为 HTTP 标头公开。	`x-example-action: something`

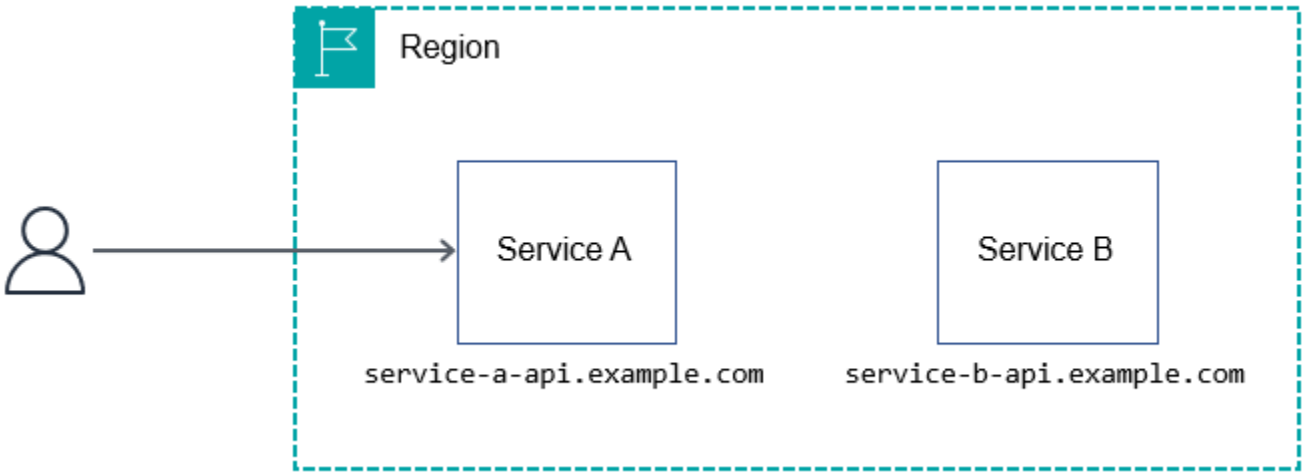
本节概述了这三种路由方法的典型用例及其需要取舍的特性，以帮助您确定哪种方法最适合您的要求和组织结构。

主机名路由模式

按主机名路由是一种通过为每个 API 提供自己的主机名来隔离 API 服务的机制；例如，`service-a.api.example.com` 或 `service-a.example.com`。

典型用例

通过使用主机名进行路由，可以减少发布中的摩擦，因为服务团队之间不共享任何内容。团队将负责管理从 DNS 条目到生产中服务操作的所有内容。



优点

对于 HTTP API 路由，主机名路由是迄今为止最直接、最具扩展性的方法。您可以使用任何相关 AWS 服务来构建遵循此方法的架构：您可以使用 Amazon API Gateway、AWS AppSync、应用程序负载均衡器和 Amazon Elastic Compute Cloud（Amazon EC2）或任何其他符合 HTTP 的服务来创建架构。

团队可以使用主机名路由，以完全拥有其自己的子域。它还可以更轻松地隔离、测试和编排特定 AWS 区域 或版本的部署；例如，`region.service-a.api.example.com` 或 `dev.region.service-a.api.example.com`。

缺点

当您使用主机名路由时，您的使用者必须记住不同的主机名，才能与您公开的每个 API 进行交互。您可以通过提供客户端开发工具包来缓解此问题。但是，客户端开发工具包有其自身的挑战。例如，他们必须支持滚动更新、多种语言、版本控制、沟通由安全问题或错误修复引起的破坏性变更、文档等。

使用主机名路由时，还需要在每次创建新服务时注册子域或域。

路径路由模式

按路径路由是一种将多个或所有 API 分组到同一个主机名下，并使用请求 URI 来隔离服务的机制；例如，`api.example.com/service-a` 或 `api.example.com/service-b`。

典型用例

大多数团队之所以选择这种方法，是因为他们想要一个简单的架构，开发者只需要记住一个 URL（例如 `api.example.com`）即可与 HTTP API 交互。API 文档通常更容易执行摘要，因为它通常是放在一起的，而不是分到不同的门户或 PDF。

对于共享 HTTP API，基于路径的路由被认为是一种简单的机制。但是，它涉及操作开销，例如配置、授权、集成，以及由于多跳而导致的额外延迟。它还需要成熟的变更管理流程，从而确保错误配置不会中断所有服务。

在 AWS 中，有多种方法可以共享 API 并有效地路由到正确的服务。以下各节讨论了三种方法：HTTP 服务反向代理、API 网关和 Amazon CloudFront。统一 API 服务的建议方法都不依赖于运行 AWS 的下游服务。只要与 HTTP 兼容，这些服务就可以在任何地点，或在任何技术上运行，而不会出现问题。

HTTP 服务反向代理

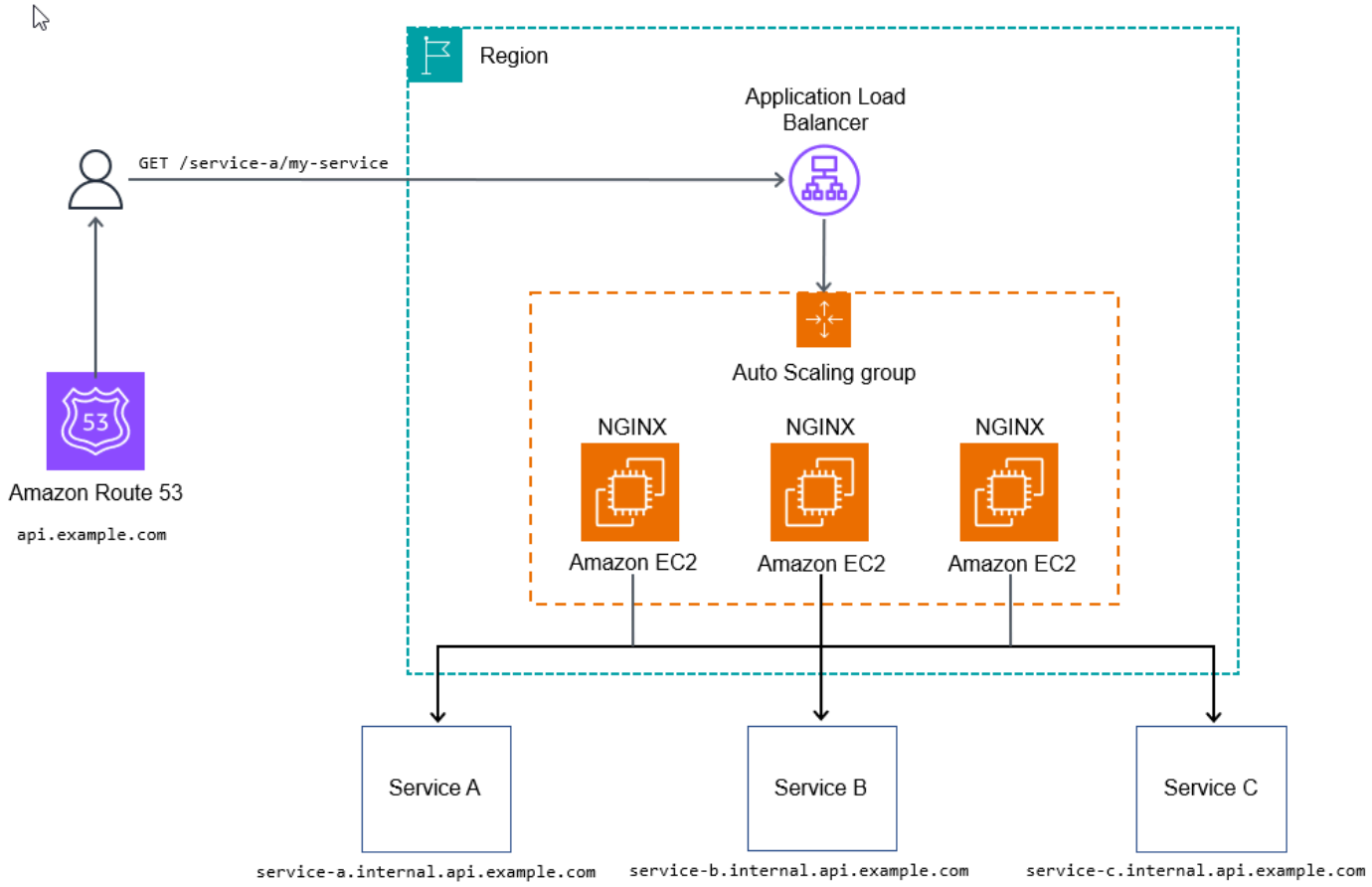
您可以使用像 NGINX 这样的 HTTP 服务器来创建动态路由配置。在 Kubernetes 架构中，您还可以创建入口规则，以匹配服务的路径。（本指南未涉及 Kubernetes 入口；有关更多信息，请参阅 Kubernetes 文档。）

以下 NGINX 配置将 `api.example.com/my-service/` 的 HTTP 请求动态映射到 `my-service.internal.api.example.com`。

```
server {
    listen  80;

    location (^/[\w-]+)/(.*) {
        proxy_pass $scheme://$1.internal.api.example.com/$2;
    }
}
```

下图说明了 HTTP 服务反向代理方法。



对于某些不使用其他配置开始处理请求，允许下游 API 收集指标和日志的用例，这种方法可能就足够了。

要为经营生产做好准备，您需要能够为堆栈的每个级别添加可观测性、添加其他配置，或者添加脚本以自定义 API 入口点，以允许更高级的功能，例如速率限制或使用令牌。

优点

HTTP 服务反向代理方法的最终目标是创建一种可扩展且可管理的方法，将 API 统一到一个域中，以便其对所有 API 使用者表现得连贯一致。这种方法还使您的服务团队能够部署和管理自己的 API，并使部署后的开销最小。用于跟踪的 AWS 托管服务（例如 AWS X-Ray 或 AWS WAF）在这里仍然适用。

缺点

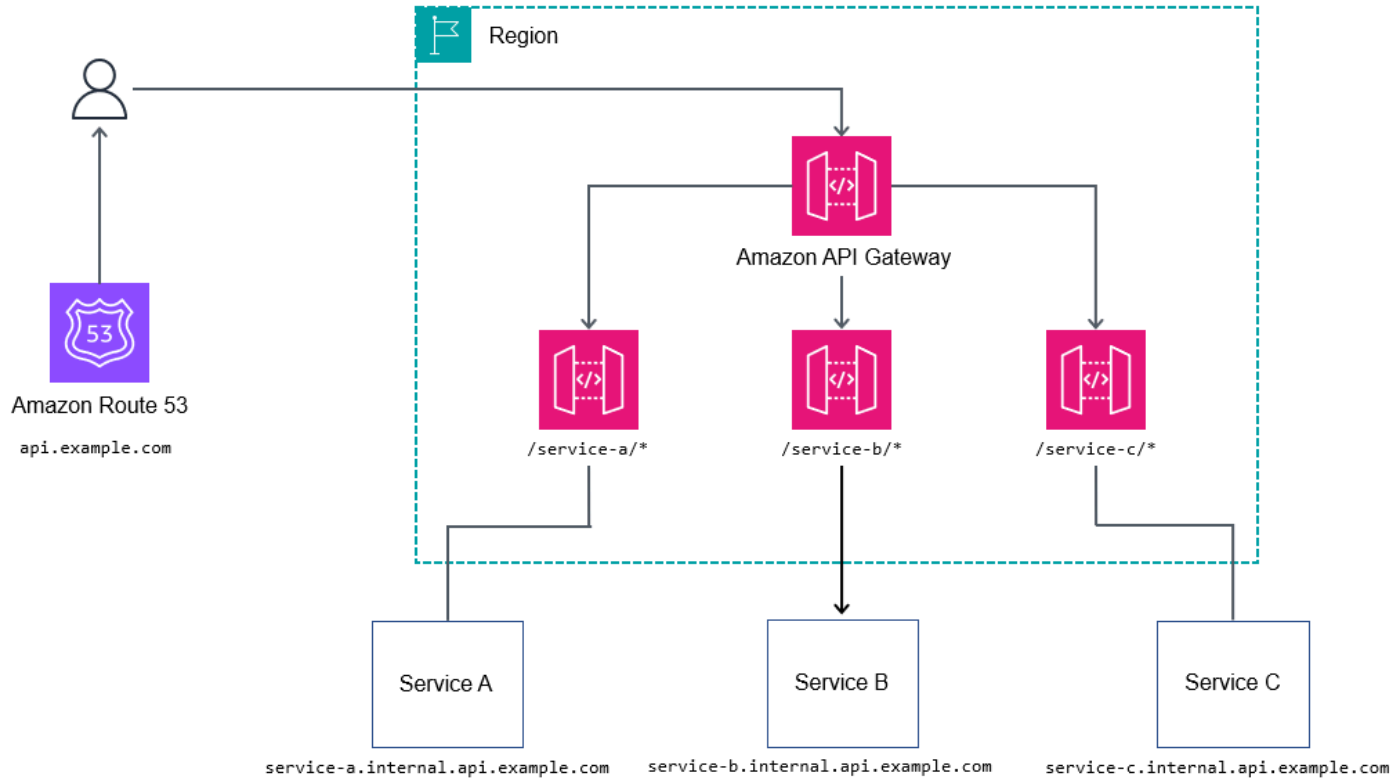
这种方法的主要缺点是需要对所需的基础设施组件进行大量的测试和管理，但如果您有站点可靠性工程（SRE）团队，这可能不是问题。

这种方法存在一个成本临界点。在中低量下，它比本指南中讨论的其他一些方法更昂贵。在大批量情况下，它非常具有成本效益（每秒约 10 万个事务或更高）。

API Gateway

Amazon API Gateway 服务（REST API 和 HTTP API）可以采用与 HTTP 服务反向代理方法类似的方式，对流量进行路由。在 HTTP 代理模式下使用 API 网关提供了一种简单的方法，可以将许多服务封装到顶级子域 `api.example.com` 的入口点中，然后将请求代理到嵌套服务；例如 `billing.internal.api.example.com`。

您可能不想在根网关或核心 API 网关中每个服务内映射每条路径，从而变得过于精细。相反，更倾向选择通配符路径，例如使用 `/billing/*` 将请求转发到计费服务。通过不映射根网关或核心 API 网关中的所有路径，您可以更为灵活地使用 API，因为您不必在每次更改 API 时都对根 API 网关进行更新。



优点

为了控制更复杂的工作流程，例如更改请求属性，REST API 将公开 Apache Velocity 模板语言（VTL），以允许您修改请求和响应。REST API 可以提供其他益处，例如：

- Auth N/Z，使用 AWS Identity and Access Management（IAM）、Amazon Cognito 或 AWS Lambda 授权方
- 用于追踪的 AWS X-Ray
- 与 AWS WAF 集成
- 基本速率限制
- 用于将使用者分组到不同等级的使用令牌（参见 API 网关文档中的限制 API 请求以获得更高的吞吐量）

缺点

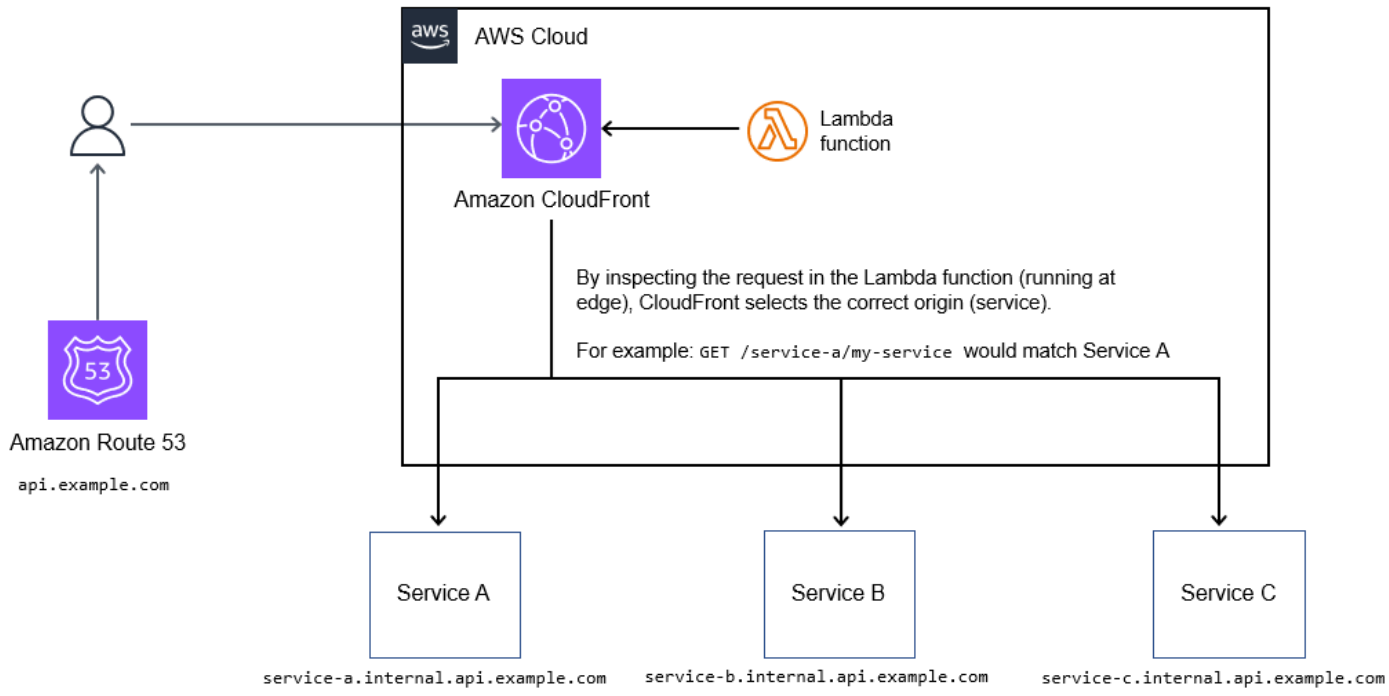
在大流量下，某些用户可能会遇到成本问题。

CloudFront

您可以使用 Amazon CloudFront 中的动态源选择功能，有条件地选择要转发请求的来源（服务）。您可以使用此功能通过单个主机名（例如 `api.example.com`）路由多个服务。

典型用例

路由逻辑以 Lambda@Edge 函数中的代码形式存在，因此它支持高度可定制的路由机制，例如 A/B 测试、金丝雀发布、功能标记和路径重写。此过程如下图所示。



优点

如果您需要缓存 API 响应，则此方法是一个很好的方法，可以将服务集合统一到单个端点之后。这是一种经济实惠的方法，可以统一 API 集合。

此外，CloudFront 还支持字段级加密，以及可实现基本速率限制和基本 ACL 的与 AWS WAF 集成。

缺点

此方法最多支持 250 个可以统一的源（服务）。此限值对于大多数部署来说已经足够，但是随着服务组合的增长，它可能会导致大量 API 出现问题。

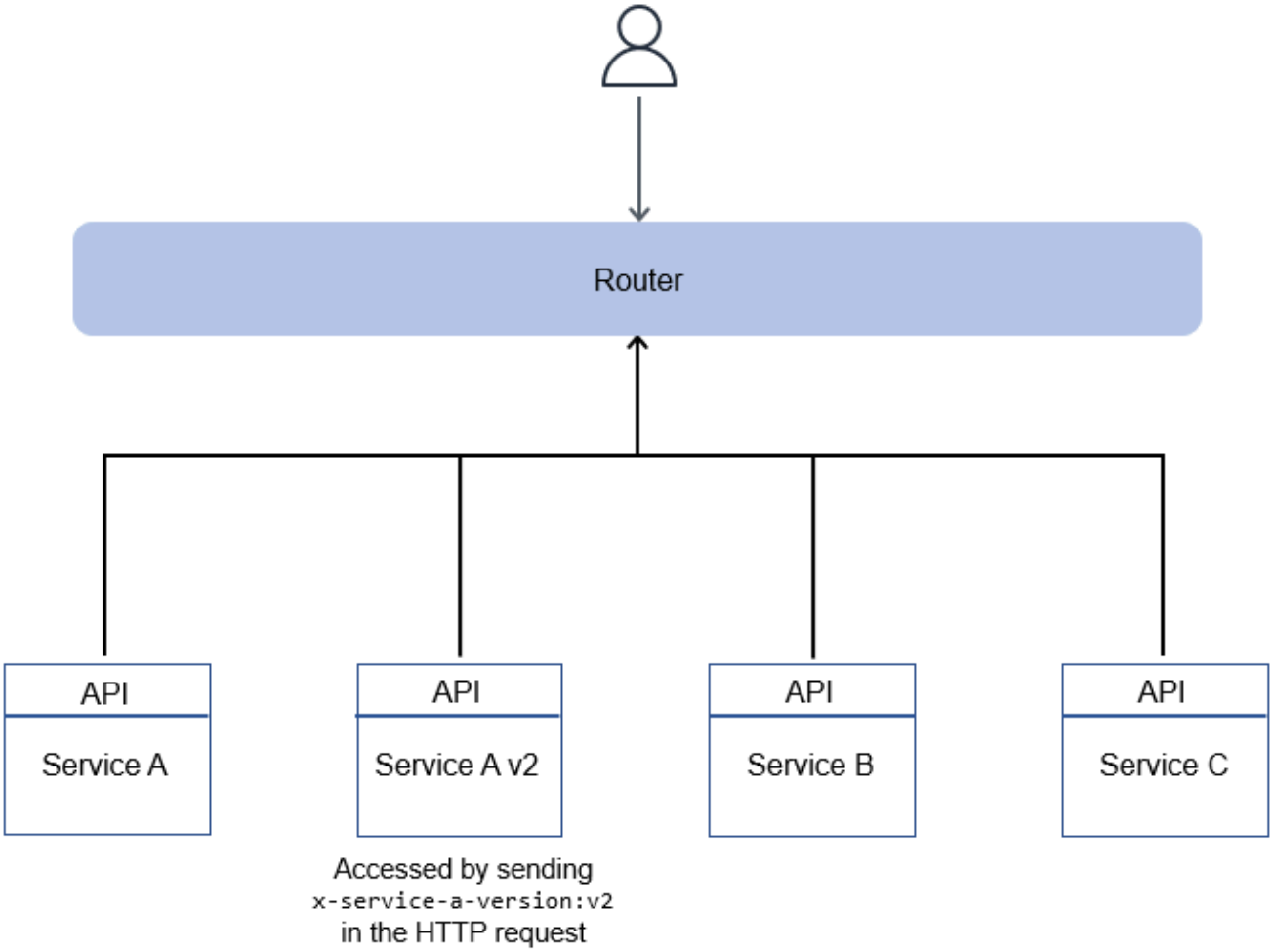
更新 Lambda@Edge 函数目前需要花费几分钟的时间。CloudFront 还需要最多 30 分钟才能将更改传播到所有节点。这最终会阻止更新完成前的进一步更新。

HTTP 标头路由模式

基于标头的路由允许您通过在 HTTP 请求中指定 HTTP 标头，为每个请求定位正确的服务。例如，发送标头 `x-service-a-action: get-thing` 将使您能够从 `Service A get thing`。请求的路径仍然很重要，因为它为您尝试使用哪种资源提供了指导。

除了将 HTTP 标头路由用于操作之外，您还可以将其用作版本路由、启用功能标志、A/B 测试或类似需求的机制。实际上，您很可能会将标头路由与其他某种路由方法共同使用来创建强大的 API。

通常，在微服务前面，HTTP 标头路由的架构有一个薄路由层，该层会路由到正确的服务并返回响应，如下图所示。该路由层可以覆盖所有服务，或仅覆盖少数服务，以实现基于版本的路由之类的操作。



优点

配置更改只需极小的工作量，而且可以轻松实现自动化。这种方法也很灵活，支持创造性的方法，可以只公开您想从服务中获得的特定操作。

缺点

与主机名路由方法一样，HTTP 标头路由假定您可以完全控制客户端，并且可以操作自定义 HTTP 标头。代理、内容分发网络（CDN）和负载均衡器可以限制标头的大小。尽管不太可能，但这仍然有机会成为一个问题，具体取决于您添加了多少标题和 cookie。

断路器模式

意图

断路器模式可以防止调用方服务在调用先前导致反复超时或失败时重试调用其他服务（被调用方）。该模式还用于检测被调用方服务何时恢复运行。

动机

多个微服务协作处理请求时，一个或多个服务可能会变得不可用或出现高延迟。复杂的应用程序使用微服务时，一个微服务中断可能会导致应用程序故障。微服务通过远程程序调用进行通信，网络连接中可能会出现瞬态错误，从而导致故障。（暂时性错误可以通过使用回退重试模式来处理。）在同步执行期间，超时或故障的级联效应可能会导致用户体验不佳。

但是，在某些情况下，故障可能需要更长的时间才能解决，例如，当被调用方服务关闭或数据库争用导致超时时。在这种情况下，如果调用服务反复重试调用，则这些重试可能会导致网络争用和数据库线程池消耗。此外，如果多个用户反复重试应用程序，则会使问题变得更糟，且可能导致整个应用程序的性能下降。

Michael Nygard 在他的《Release It》（Nygard 2018）一书中推广了断路器模式。此设计模式可以防止调用方服务重试先前导致反复超时或失败的服务调用。它还可以检测被调用方服务何时恢复运行。

断路器对象的工作原理类似于电路断路器，当电路出现异常时，断路器会自动切断电流。电路出现故障时，电路断路器会切断电流，也就是跳闸。同样，断路器对象位于调用方和被调用方服务之间，如果被调用方不可用，则断路器会跳闸。

分布式计算的谬误是 Sun Microsystems 的 Peter Deutsch 等人提出的一系列断言。他们说，刚接触分布式应用程序的程序员总是会做出错误的假设。网络可靠性、零延迟期望和带宽限制导致软件应用程序在编写时对网络错误几乎没有错误处理。

在网络中断期间，应用程序可能会无限期地等待回复并持续消耗应用程序资源。网络可用时未能重试这些操作也可能导致应用程序性能下降。如果对数据库或外部服务的 API 调用由于网络问题而超时，则在没有断路器的情况下重复调用可能会影响成本和性能。

适用性

在以下情况下使用此模式：

- 调用方服务发起了一个极有可能失败的调用。

- 被调用方服务表现出的高延迟（例如，当数据库连接速度较慢时）会导致调用方服务超时。
- 调用方服务发起同步调用，但被调用方服务不可用或延迟很长。

问题和注意事项

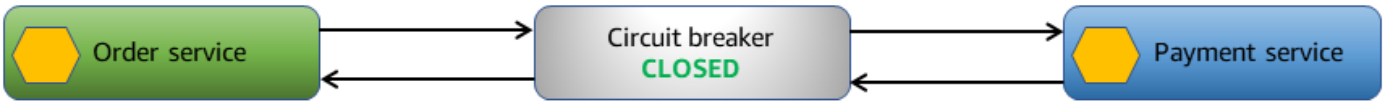
- 与服务无关的实现：为防止代码膨胀，我们建议您以与微服务无关且由 API 驱动的方式实现断路器对象。
- 由被调用方关闭电路：当被调用方从性能问题或故障中恢复时，他们可以将电路状态更新为 CLOSED。这是断路器模式的扩展，如果您的恢复时间目标（RTO）需要，则可以实施。
- 多线程调用：到期超时值定义为在再次路由调用以检查服务可用性之前，电路保持跳闸的时间段。在多个线程中调用被调用者服务时，第一个失败的调用将定义到期超时值。您的实现应确保后续调用不会无休止地更改到期超时时间。
- 强制打开或关闭电路：系统管理员应该能够打开或关闭电路。这可以通过更新数据库表中的到期超时值来完成。
- 可观测性：应用程序应设置日志记录，以识别断路器打开时失败的调用。

实施

高级架构

在以下示例中，调用方是订单服务，被调用方是付款服务。

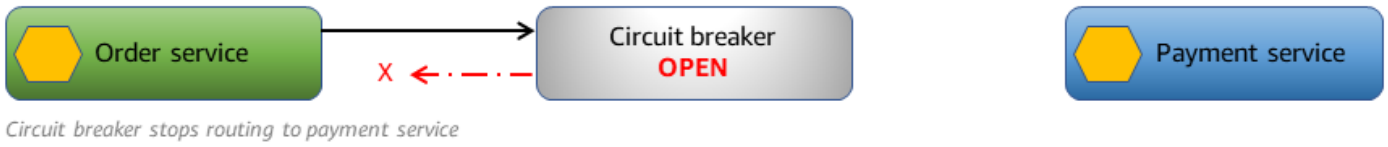
如果没有故障，订单服务会通过断路器将所有调用路由到付款服务，如下图所示。



如果付款服务超时，断路器可以检测到超时并跟踪故障。

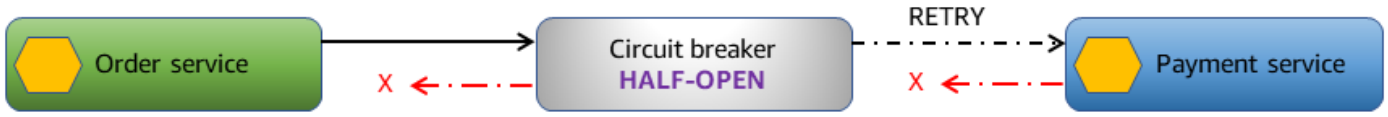
Circuit breaker with payment service failure

如果超时超过指定的阈值，则应用程序将打开电路。当电路开启时，断路器对象不会将调用路由到付款服务。当订单服务调用付款服务时，会立即返回失败结果。



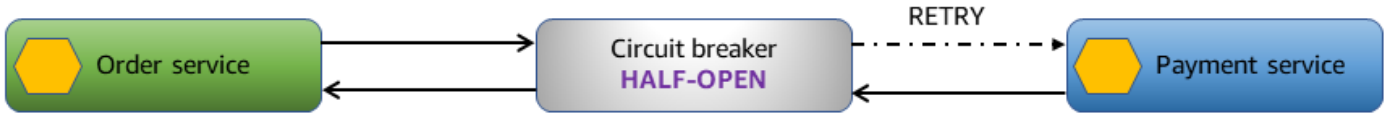
Circuit breaker stops routing to payment service

断路器对象会定期尝试查看对付款服务的调用是否成功。



Circuit breaker periodically retries payment service

成功调用付款服务后，电路将关闭，所有后续调用将再次路由到付款服务。



Circuit breaker with working payment service

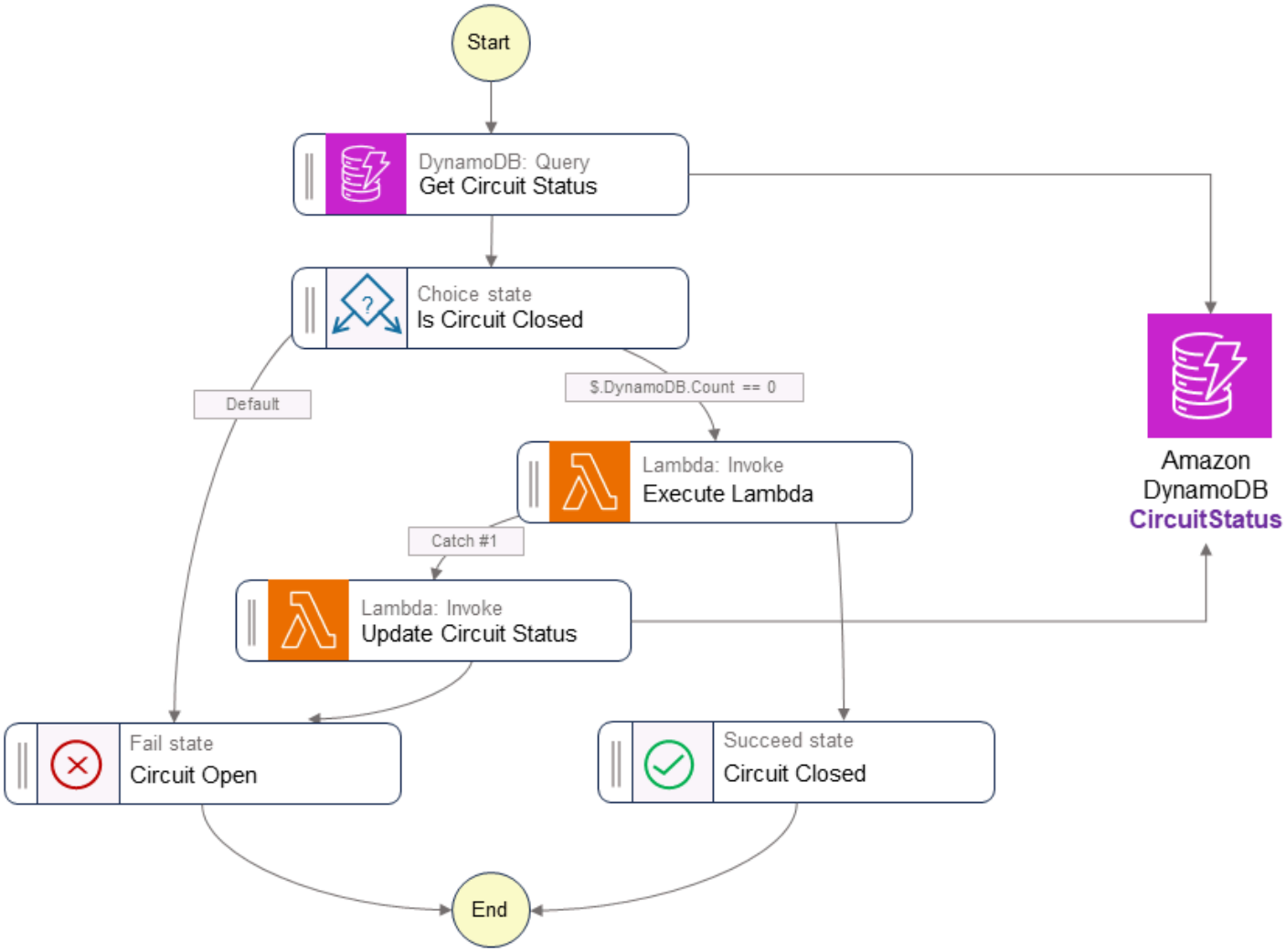
使用 AWS 服务实施

示例解决方案使用 AWS Step Functions 中的快速工作流程，来实现断路器模式。Step Functions 状态机允许您配置模式实现所需的重试功能和基于决策的控制流。

该解决方案还使用 Amazon DynamoDB 表作为数据存储来跟踪电路状态。为了提高性能，可以将其替换为诸如亚马逊 ElastiCache （Redis OSS）之类的内存数据存储。

当服务想要调用其他服务时，它会使用被调用服务的名称启动工作流程。该工作流程从 DynamoDB `CircuitStatus` 表中获取断路器状态，该表存储了当前已降级的服务。如果 `CircuitStatus` 包含被调用方的未过期记录，则电路处于打开状态。Step Functions 工作流程立即返回失败并以 `FAIL` 状态退出。

如果 `CircuitStatus` 表不包含被调用方的记录或包含过期的记录，则表示服务正常运行。状态机定义中的 `ExecuteLambda` 步骤调用通过参数值发送的 Lambda 函数。如果调用成功，Step Functions 工作流程将以 `SUCCESS` 状态退出。



如果服务调用失败或出现超时，应用程序将在规定的次数内以指数回退方式重试。如果重试后服务调用失败，则工作流程会在 CircuitStatus 表中为该服务插入一条记录，值为 ExpiryTimeStamp，且工作流程以 FAIL 状态退出。只要断路器处于打开状态，随后对同一服务的调用就会立即返回故障结果。状态机定义中的 Get Circuit Status 步骤根据 ExpiryTimeStamp 值检查服务可用性。使用 DynamoDB 生存时间（TTL）功能将过期项目从 CircuitStatus 表中删除。

代码示例

以下代码使用 GetCircuitStatus Lambda 函数检查断路器状态。

```
var serviceDetails = _dbContext.QueryAsync<CircuitBreaker>(serviceName,
 QueryOperator.GreaterThan,
                new List<object>
                    {currentTimeStamp}).GetRemainingAsync();
```

```
if (serviceDetails.Result.Count > 0)
{
    functionData.CircuitStatus = serviceDetails.Result[0].CircuitStatus;
}
else
{
    functionData.CircuitStatus = "";
}
```

以下代码显示了 Step Functions 工作流程中的 Amazon States Language 语句。

```
"Is Circuit Closed": {
    "Type": "Choice",
    "Choices": [
    {
        "Variable": "$.CircuitStatus",
        "StringEquals": "OPEN",
        "Next": "Circuit Open"
    },
    {
        "Variable": "$.CircuitStatus",
        "StringEquals": "",
        "Next": "Execute Lambda"
    }
    ]
},
"Circuit Open": {
    "Type": "Fail"
}
```

GitHub 存储库

有关此模式示例架构的完整实现，请参见 GitHub 存储库，网址为https://github.com/aws-samples/circuit-breaker-netcore-blog。

参考博客文章

- 将断路器模式与 AWS Step Functions 和 Amazon DynamoDB 一起使用

相关内容

- strangler fig 模式
- 使用退避模式重试

事件溯源模式

意图

在事件驱动架构中，事件溯源模式将导致状态更改的事件存储在数据存储中。这有助于捕获和维护有关状态变化的完整历史记录，并可提高可审计性、可追溯性和分析过去状态的能力。

动机

多个微服务可以协作处理请求，并且它们通过事件进行通信。这些事件可能导致状态（数据）更改。按照事件对象的出现顺序来存储可以提供有关数据实体当前状态的宝贵信息，以及有关数据实体如何到达该状态的其他信息。

适用性

在以下情况下使用事件溯源模式：

- 跟踪需要应用程序中发生事件的不可变历史记录。
- 多语言数据预测需要来自单一事实来源（SSOT）。
- 需要对应用程序状态进行时间点重构。
- 不需要长期存储应用程序状态，但您可能需要根据需要对其进行重构。
- 工作负载具有不同的读取和写入卷。例如，您有不需要实时处理的写入密集型工作负载。
- 需要更改数据捕获（CDC）来分析应用程序性能和其他指标。
- 出于报告和合规目的，系统中发生的所有事件都需要审计数据。
- 您想通过在重播过程中更改（插入、更新或删除）事件来派生假设场景，进而确定可能的结束状态。

问题和注意事项

- 乐观并发控制：此模式存储导致系统状态更改的每个事件。多个用户或服务可能尝试同时更新同一条数据，从而导致事件冲突。当同时创建和应用冲突事件时，便会发生这些冲突，从而导致最终的数据状态与现实不符。要解决此问题，您可以实施策略来检测事件冲突并解决冲突。例如，您可以通过纳入版本控制或向事件添加时间戳来跟踪更新顺序，从而实现乐观的并发控制方案。

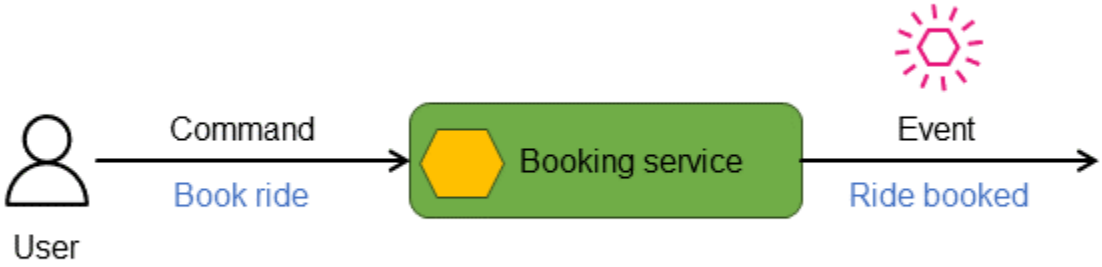
- 复杂度：实施事件溯源需要将思维方式从传统的 CRUD 运营转变为事件驱动的思维。用于将系统恢复到其原始状态的重播过程可能很复杂，用以确保数据的幂等性。事件存储、备份和快照也会增加复杂度。
- 最终一致性：由于使用命令查询责任分割（CQRS）模式或实体化视图更新数据会出现延迟，因此从事件中派生的数据预测是最终一致的。当使用者处理来自事件存储的数据且发布者发送新数据时，数据投影或应用程序对象可能无法代表当前状态。
- 查询：与传统数据库相比，从事件日志中检索当前数据或聚合数据可能更复杂、更耗时，对于复杂的查询和报告任务尤其如此。为了缓解此问题，通常使用 CQRS 模式实现事件溯源。
- 事件存储的大小和成本：随着事件的持续持久化，事件存储的大小可能会呈指数级增长，尤其是在事件吞吐量高或保留期较长的系统中。因此，您必须定期将事件数据归档到经济高效的存储中，以防止事件存储变得过大。
- 事件存储的可扩展性：事件存储必须同时有效地处理大量的写入和读取操作。扩展事件存储可能具有挑战性，因此拥有可提供分片和分区的数据存储非常重要。
- 效率和优化：选择或设计可高效处理写入和读取操作的事件存储。应针对应用程序的预期事件量和查询模式，优化事件存储。在重构应用程序状态时，实施索引和查询机制可以加快事件的检索速度。您也可以考虑使用提供查询优化功能的专门事件存储数据库或库。
- 快照：您必须通过基于时间的激活定期对事件日志进行备份。重播上次已知成功备份数据时的事件，应该会导致应用程序状态的时间点恢复。恢复点目标（RPO）是指自上一个数据恢复点以来可接受的最长时间。RPO 决定了从上一个恢复点到服务中断之间，可接受的数据丢失情况。数据和事件存储每日快照的频率应基于应用程序的 RPO 确定。
- 时间敏感性：按其发生顺序对事件进行存储。因此，在实施此模式时，网络可靠性是需要考虑的重要因素之一。延迟问题可能导致不正确的系统状态。使用最多一次的先入先出（FIFO）队列将事件传送到事件存储。
- 事件重播性能：重播大量事件以重构当前应用程序状态，可能很耗时。需要进行优化以提高性能，尤其是在重播归档数据中的事件时。
- 外部系统更新：使用事件溯源模式的应用程序可能会更新外部系统中的数据存储，并可能将这些更新捕获为事件对象。在事件重播期间，如果外部系统预计不会有更新，则这种情况可能会成为问题。在这种情况下，您可以使用功能标志来控制外部系统的更新。
- 外部系统查询：当外部系统调用对调用的日期和时间敏感时，可以将接收到的数据存储在内部数据存储中，以便在重播期间使用。
- 事件版本控制：随着应用程序的发展，事件（架构）的结构可能更改。需要对事件实施版本控制策略，以确保向后和向前兼容。这可能包括在事件有效负载中包含版本字段，并在重播期间适当地处理不同的事件版本。

实现

高级架构

命令和事件

在分布式、事件驱动的微服务应用程序中，命令代表发送至服务的指令或请求，其目的通常是启动其状态的更改。该服务会处理这些命令，并评估命令的有效性和对当前状态的适用性。如果命令成功运行，则该服务会发出一个表示所采取的操作和相关状态信息的事件，进行响应。例如，在下图中，预订服务通过发出“乘车已预订”事件来响应“预定乘车”命令。



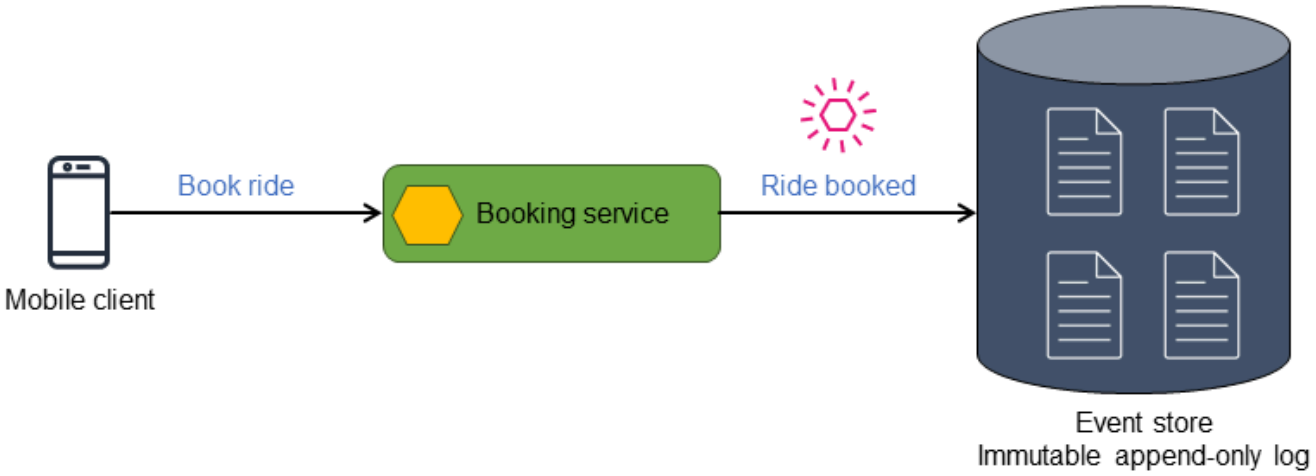
事件存储

事件记录到不可变、仅追加、按时间顺序排列的存储库，或称为事件存储的数据存储中。每个状态更改都会被视为一个单独的事件对象。通过按事件发生顺序重播事件，可以重构具有已知初始状态、当前状态和任何时间点视图的实体对象或数据存储。

事件存储充当所有操作和状态变化的历史记录，也是宝贵的单一事实来源。您可以使用事件存储通过重播处理器传递事件，进而派生系统的最终最新状态；重播处理器应用这些事件来生成最新系统状态的准确表示。您还可以使用事件存储通过重播处理器重播事件，从而生成状态的时间点视角。在事件溯源模式中，最新事件对象可能无法完全代表当前状态。您可以通过以下三种方式之一派生当前状态：

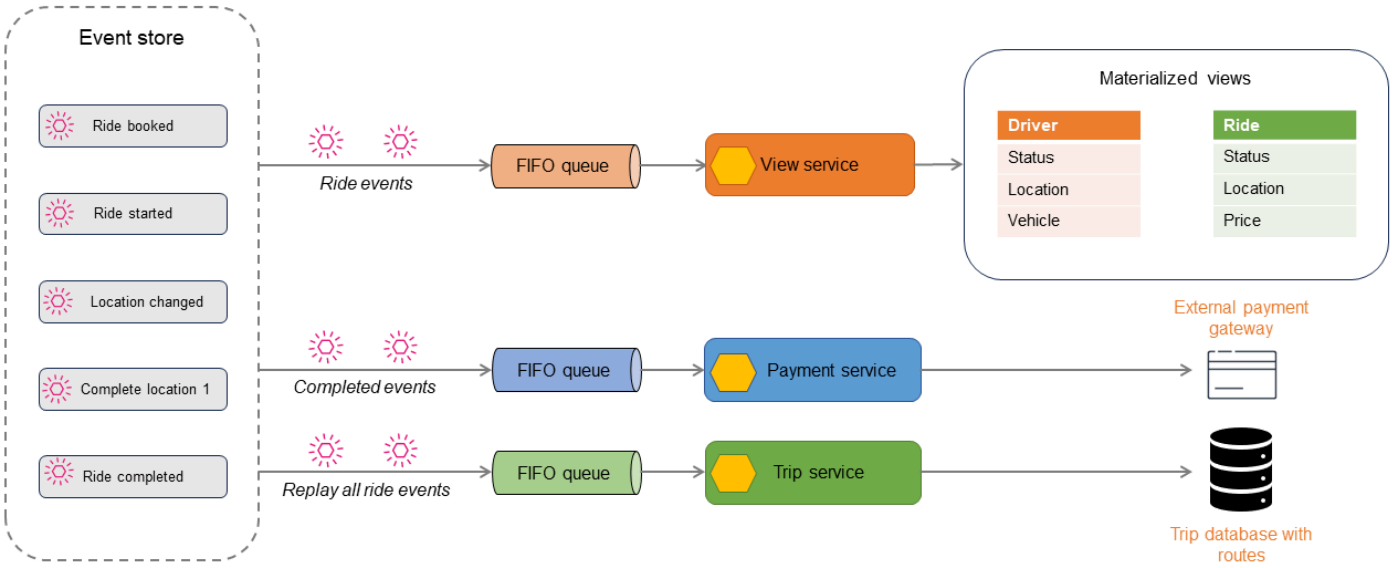
- 通过汇总相关事件。将相关的事件对象组合起来生成当前状态以供查询。这种方法通常与 CQRS 模式结合使用，因为事件被合并并写入只读数据存储中。
- 通过使用实体化视图。您可以使用带有实体化视图模式的事件溯源来计算或汇总事件数据，并获取相关数据的当前状态。
- 通过重播事件。可以重播事件对象，以执行生成当前状态的操作。

下图显示了存储在事件存储中的 `Ride booked` 事件。



事件存储发布其存储的事件，然后可以对事件进行筛选并路由到相应的处理器以进行后续操作。例如，可以将事件路由到视图处理器，该处理器汇总状态并显示实体化视图。将事件转换为目标数据存储的数据格式。可以将对这种架构进行扩展，以派生不同类型的数据存储，从而实现数据的多语言持久性。

下图介绍了乘车预订应用程序中的事件。应用程序内发生的所有事件都会存储在事件存储中。然后，对存储的事件进行筛选，并路由至不同的使用者。



通过使用 CQRS 或实体化视图模式，乘车事件可用于生成只读数据存储。您可以通过查询读取存储来获取乘车、司机或预订的当前状态。某些事件（例如 `Location changed` 或 `Ride completed`）会

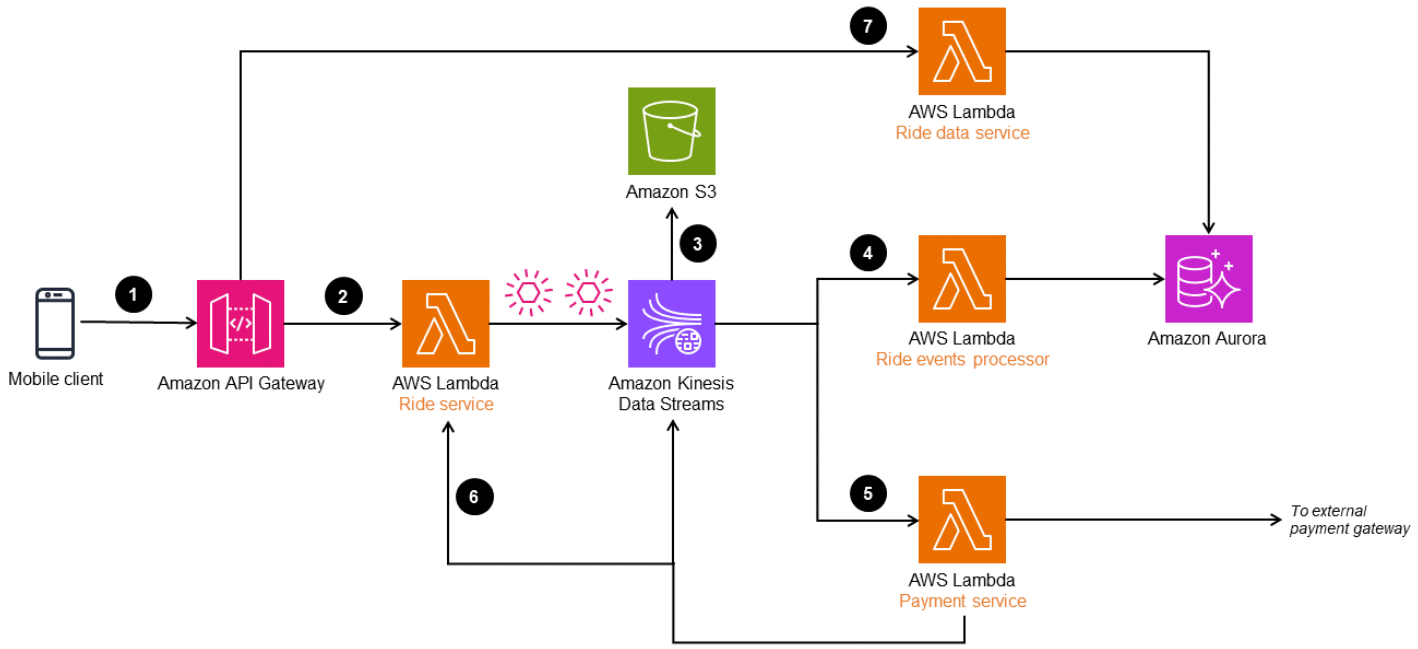
发布给其他使用者进行付款处理。乘车完成后，将重播所有乘车事件，以建立乘车历史记录，供审计或报告之用。

事件溯源模式通常用于需要时间点恢复的应用程序，也经常用于必须使用单一事实来源以不同的格式投影数据时。这两种操作都需要重播过程来运行事件并派生所需的结束状态。重播处理器可能还需要一个已知的起点，理想情况下不需要从应用程序启动开始，因为那不够高效。建议您定期拍摄系统状态的快照，并应用较少的事件来派生最新的状态。

使用亚马逊云科技服务来实施

在以下架构中，将 Amazon Kinesis Data Streams 用作事件存储。该服务将应用程序更改作为事件，对其进行捕获和管理，并提供高吞吐量和实时数据流解决方案。要在亚马逊云科技云上实施事件溯源模式，您还可以根据应用程序的需求使用诸如 Amazon EventBridge 和 Amazon Managed Streaming for Apache Kafka（Amazon MSK）等服务。

为了增强持久性并支持审计，您可以将 Kinesis Data Streams 捕获的事件归档到 Amazon Simple Storage Service（Amazon S3）中。此双存储方法有助于安全地保留历史事件数据，以备将来分析和合规之用。



工作流包含以下步骤：

1. 乘车预订请求是通过移动客户端向 Amazon API Gateway 端点提出的。
2. 乘车微服务（`Ride service` Lambda 函数）接收请求，转换对象，然后发布到 Kinesis Data Streams。

3. 将 Kinesis Data Streams 中的事件数据存储在 Amazon S3 中，以用于合规和审计历史记录。
4. 这些事件由 `Ride event processor` Lambda 函数转换和处理，并存储在 Amazon Aurora 数据库中，为乘车数据提供实体化视图。
5. 已完成的乘车事件会被筛选并发送到外部支付网关进行付款处理。付款完成后，将向 Kinesis Data Streams 发送另一个事件以更新乘车数据库。
6. 乘车完成后，乘车事件将重播到 `Ride service` Lambda 函数中，以构建路线和乘车历史记录。
7. 乘车信息可通过 `Ride data service` 读取，该服务从 Aurora 数据库读取信息。

API Gateway 还可以在无需 `Ride service` Lambda 函数的情况下，将事件对象直接发送到 Kinesis Data Streams。但是，在叫车服务等复杂系统中，可能需要对事件对象进行处理和丰富，然后才能将其摄入数据流。出于这个原因，该架构具有在将事件发送到 Kinesis Data Streams 之前对其进行处理的 `Ride service`。

参考博客文章

- Amazon Lambda 的新增功能 – 作为事件源的 SQS FIFO

六边形架构模式

意图

六边形架构模式（也称为端口和适配器模式）由 Alistair Cockburn 博士在 2005 年提出。它旨在创建松散耦合的架构，在这种架构中，可以独立测试应用程序组件，而不依赖数据存储或用户界面（UIs）。这种模式有助于防止数据存储被技术锁定，并且 UIs. 这样一来，随着时间的推移，技术堆栈的更改就变得更加容易，对业务逻辑的影响也有限或没有影响。在此松耦合架构中，应用程序通过称为端口的接口与外部组件通信，并使用适配器来转换与这些组件的技术交互。

动机

六边形架构模式用于将业务逻辑（域逻辑）与相关的基础架构代码（例如用于访问数据库或外部的代码）隔离开来。 APIs这种模式对于为需要与外部服务集成的 AWS Lambda 函数创建松散耦合的业务逻辑和基础架构代码非常有用。在传统架构中，常见的做法是将业务逻辑作为存储过程嵌入数据库层，并嵌入到用户界面中。此做法再加上在业务逻辑内使用特定于用户界面的结构，会导致紧密耦合的架构，从而在数据库迁移和用户体验（UX）现代化工作中造成瓶颈。六边形架构模式使您能够按目的（而不是按技术），来设计系统和应用程序。此策略可以生成易于交换的应用程序组件，例如数据库、UX 和服务组件。

适用性

在以下情况下使用六边形架构模式：

- 您想解耦应用程序架构，以创建可以全面测试的组件。
- 多种类型的客户端可以使用相同的域逻辑。
- 您的用户界面和数据库组件需要定期进行技术更新，而不会影响应用程序逻辑。
- 您的应用程序需要多个输入提供程序和输出使用者，而自定义应用程序逻辑会导致代码复杂和缺乏可扩展性。

问题和注意事项

- 领域驱动型设计：六边形架构特别适用于领域驱动设计（DDD）。每个应用程序组件都代表 DDD 中的一个子域，六边形架构可用于实现应用程序组件之间的松耦合。

- 可测试性：根据设计，六边形架构使用抽象作为输入和输出。因此，由于固有的松耦合，编写单元测试和单独测试变得更加容易。
- 复杂性：将业务逻辑与基础设施代码隔离的复杂性，如果处理得当，可以带来巨大的好处，例如敏捷性、测试覆盖率和技术适应性。否则，问题可能会变得复杂而难以解决。
- 维护开销：只有当应用程序组件需要多个输入源和输出目标进行写入时，或者输入和输出数据存储必须随着时间的推移而发生变化时，才有必要添加使架构可插拔的适配器代码。否则，适配器将成为另一个需要维护的额外层，这会带来维护开销。
- 延迟问题：使用端口和适配器会增加一层，这可能会导致延迟。

实施

Hexagonal 架构支持将应用程序和业务逻辑与基础设施代码以及将应用程序与外部 UIs APIs、数据库和消息代理集成的代码隔离开来。您可以通过端口和适配器，轻松地将业务逻辑组件连接到应用程序架构中的其他组件（例如数据库）。

端口是应用程序组件中与技术无关的入口点。这些自定义接口决定了允许外部参与者与应用程序组件通信的接口，而不管是谁或什么实现了该接口。这与 USB 端口允许许多不同类型的设备与计算机通信的原理类似，只要它们使用 USB 适配器即可。

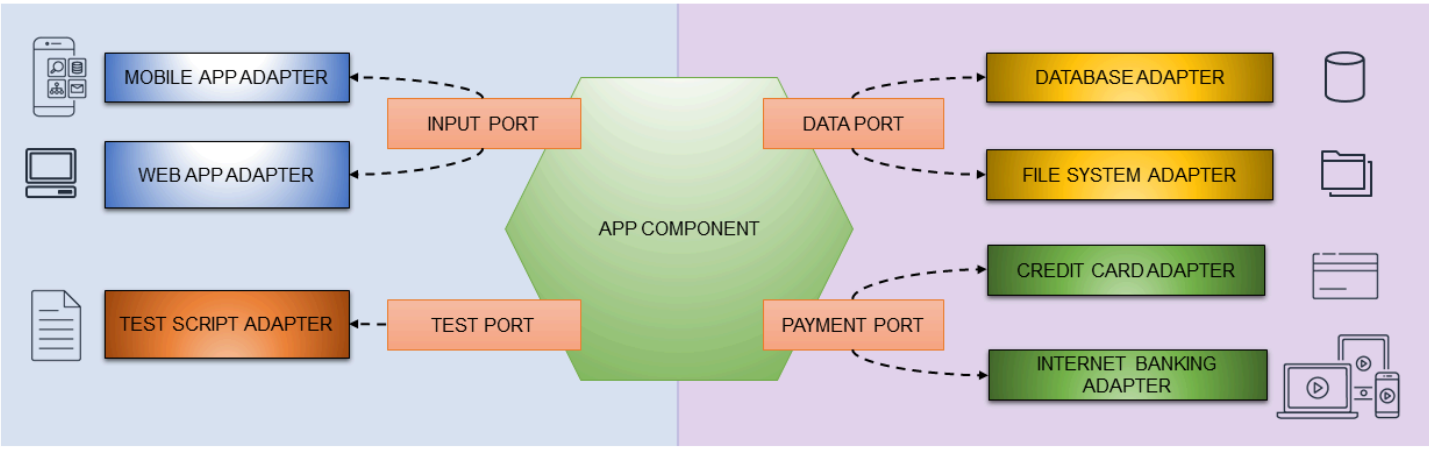
适配器使用特定技术通过端口与应用程序进行交互。适配器插入这些端口，从端口接收数据或向端口提供数据，然后转换数据以进行进一步处理。例如，REST 适配器使参与者能够通过 REST API 与应用程序组件进行通信。一个端口可以具有多个适配器，而不会对端口或应用程序组件造成任何风险。为了扩展上一个示例，向同一端口添加 GraphQL 适配器为参与者提供了一种额外的方式，可通过 GraphQL API 与应用程序交互，而不会影响 REST API、端口或应用程序。

端口连接到应用程序，适配器用作与外界的连接。您可以使用端口创建松耦合的应用程序组件，并通过更改适配器来交换依赖组件。这使应用程序组件无需任何上下文感知即可与外部输入和输出进行交互。组件可在任何级别进行互换，有助于自动化测试。您可以独立测试组件，而无需依赖基础设施代码，也无需预调配整个环境来进行测试。应用程序逻辑不依赖于外部因素，因此测试得以简化，模拟依赖关系也变得更加容易。

例如，在松耦合架构中，应用程序组件应该能够在不知道数据存储详细信息的情况下读取和写入数据。应用程序组件的职责是向接口（端口）提供数据。适配器定义写入数据存储的逻辑，数据存储可以是数据库、文件系统或对象存储系统（例如 Amazon S3），具体取决于应用程序的需求。

架构简析

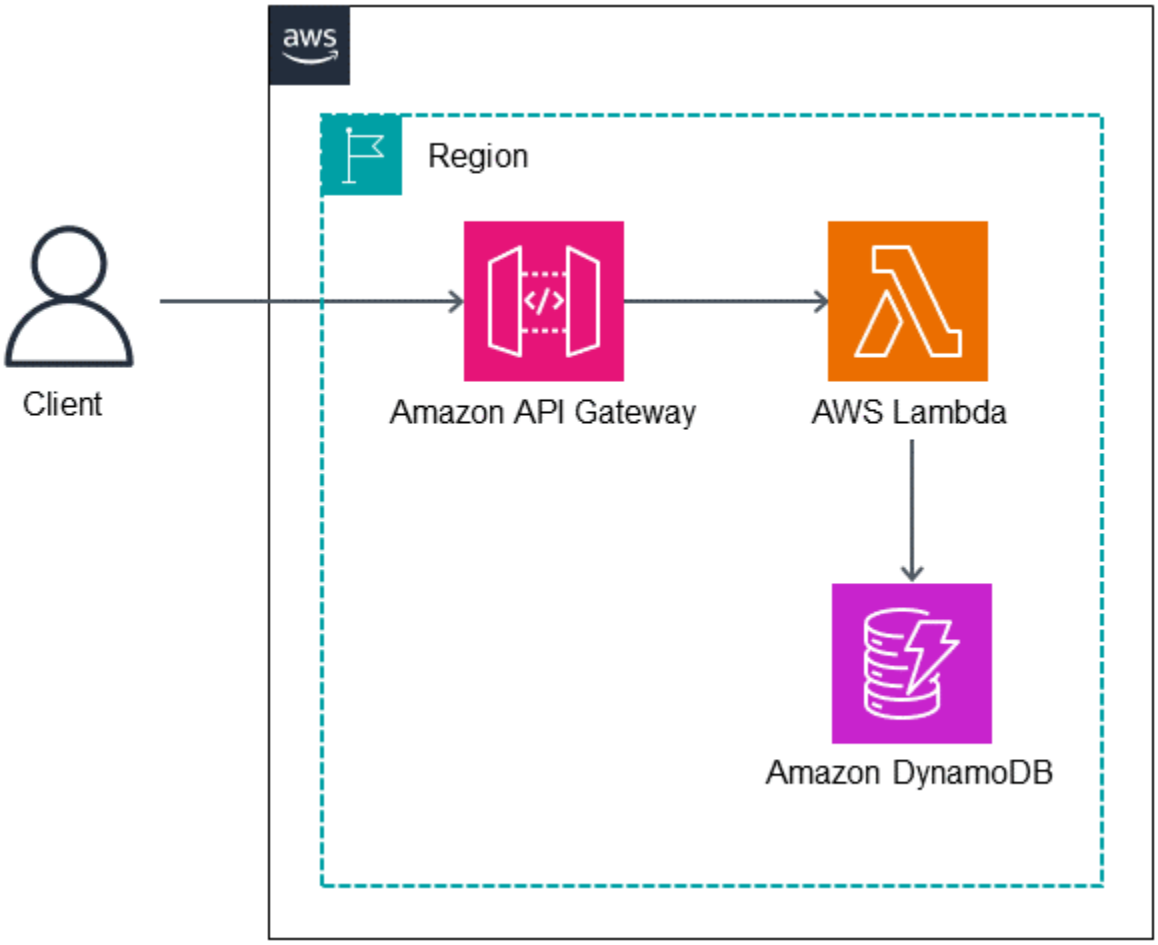
应用程序或应用程序组件包含核心业务逻辑。它从端口接收命令或查询，并通过端口向外部参与者发送请求，这些请求是通过适配器实现的，如下图所示。



使用实现方式 AWS 服务

AWS Lambda 函数通常包含业务逻辑和数据库集成代码，它们紧密耦合以实现目标。您可以使用六边形架构模式将业务逻辑与基础设施代码隔离。此隔离使得业务逻辑可以进行单元测试，而无需依赖数据库代码，从而提高了开发过程的敏捷性。

在以下架构中，Lambda 函数实现六边形架构模式。Lambda 函数由 Amazon API Gateway REST API 发起。该函数实现业务逻辑并将数据写入 DynamoDB 表。



代码示例

本节中的示例代码展示了如何使用 Lambda 实现域模型，将其与基础设施代码（例如用于访问 DynamoDB 的代码）隔离，并为该函数实现单元测试。

域模型

域模型类别对外部组件或依赖关系一无所知，它只实现业务逻辑。在以下示例中，类别 `Recipient` 是一个域模型类别，用于检查预留日期是否存在重叠。

```
class Recipient:
    def __init__(self, recipient_id:str, email:str, first_name:str, last_name:str,
 age:int):
        self.__recipient_id = recipient_id
        self.__email = email
        self.__first_name = first_name
        self.__last_name = last_name
        self.__age = age
```

```
        self.__slots = []

    @property
    def recipient_id(self):
        return self.__recipient_id
    #.....

    def are_slots_same_date(self, slot:Slot) -> bool:
        for selfslot in self.__slots:
            if selfslot.reservation_date == slot.reservation_date:
                return True
        return False

    def is_slot_counts_equal_or_over_two(self) -> bool:
    #.....
```

输入端口

RecipientInputPort 类别连接到接收者类别并运行域逻辑。

```
class RecipientInputPort(IRecipientInputPort):
    def __init__(self, recipient_output_port: IRecipientOutputPort, slot_output_port:
 ISlotOutputPort):
        self.__recipient_output_port = recipient_output_port
        self.__slot_output_port = slot_output_port

    '''
    make reservation: adapting domain model business logic
    '''
    def make_reservation(self, recipient_id:str, slot_id:str) -> Status:
        status = None

        # -----------------------------------------------------
        # get an instance from output port
        # -----------------------------------------------------
        recipient = self.__recipient_output_port.get_recipient_by_id(recipient_id)
        slot = self.__slot_output_port.get_slot_by_id(slot_id)

        if recipient == None or slot == None:
            return Status(400, "Request instance is not found. Something wrong!")

        print(f"recipient: {recipient.first_name}, slot date: {slot.reservation_date}")
```

```
        # ------------------------------------------------
        # execute domain logic
        # ------------------------------------------------
        ret = recipient.add_reserve_slot(slot)

        # ------------------------------------------------
        # persistent an instance throgh output port
        # ------------------------------------------------
        if ret == True:
            ret = self.__recipient_output_port.add_reservation(recipient)

        if ret == True:
            status = Status(200, "The recipient's reservation is added.")
        else:
            status = Status(200, "The recipient's reservation is NOT added!")
        return status
```

DynamoDB 适配器类别

DDBRecipientAdapter 类别实现对 DynamoDB 表的访问。

```
class DDBRecipientAdapter(IRecipientAdapter):
    def __init__(self):
        ddb = boto3.resource('dynamodb')
        self.__table = ddb.Table(table_name)

    def load(self, recipient_id:str) -> Recipient:
        try:
            response = self.__table.get_item(
                Key={'pk': pk_prefix + recipient_id})
        ...

    def save(self, recipient:Recipient) -> bool:
        try:
            item = {
                "pk": pk_prefix + recipient.recipient_id,
                "email": recipient.email,
                "first_name": recipient.first_name,
                "last_name": recipient.last_name,
                "age": recipient.age,
                "slots": []
            }
          # ...
```

Lambda 函数 get_recipient_input_port 是 RecipientInputPort 类别实例的工厂。它使用相关的适配器实例构造输出端口类别的实例。

```
def get_recipient_input_port():
    return RecipientInputPort(
        RecipientOutputPort(DDBRecipientAdapter()),
        SlotOutputPort(DDBSlotAdapter()))

def lambda_handler(event, context):

    body = json.loads(event['body'])
    recipient_id = body['recipient_id']
    slot_id = body['slot_id']

    # get an input port instance
    recipient_input_port = get_recipient_input_port()
    status = recipient_input_port.make_reservation(recipient_id, slot_id)

    return {
        "statusCode": status.status_code,
        "body": json.dumps({
            "message": status.message
        }),
    }
```

单元测试

您可以通过注入模拟类别来测试域模型类别的业务逻辑。以下示例提供了域模型 Recipent 类别的单元测试。

```
def test_add_slot_one(fixture_recipient, fixture_slot):
    slot = fixture_slot
    target = fixture_recipient
    target.add_reserve_slot(slot)
    assert slot != None
    assert target != None
    assert 1 == len(target.slots)
    assert slot.slot_id == target.slots[0].slot_id
    assert slot.reservation_date == target.slots[0].reservation_date
    assert slot.location == target.slots[0].location
    assert False == target.slots[0].is_vacant

def test_add_slot_two(fixture_recipient, fixture_slot, fixture_slot_2):
```

```
    #.....

def test_cannot_append_slot_more_than_two(fixture_recipient, fixture_slot,
 fixture_slot_2, fixture_slot_3):
    #.....

def test_cannot_append_same_date_slot(fixture_recipient, fixture_slot):
    #.....
```

GitHub 存储库

有关此模式示例架构的完整实现，请参阅 https://github.com/aws-samples/aws-lambda-domain-model-sample 中的 GitHub 存储库。

相关内容

- Hexagonal architecture（Alistair Cockburn 的文章）
- 使用（日语AWS 博客文章 AWS Lambda）开发进化架构

视频

以下视频（日语）讨论了如何使用 Lambda 函数在实现域模型时使用六边形架构。

发布-订阅模式

意图

发布-订阅模式，也称为“发-订”模式，是一种消息发送方（发布者）与感兴趣的接收方（订阅用户）解耦的消息模式。此模式通过称为消息代理或路由器（消息基础实施）的中介发布消息或事件，从而实现异步通信。发布-订阅模式将消息传递的责任转移给消息基础实施，从而提高了发送方的可扩展性和响应能力，由此发送方可以专注于核心消息处理。

动机

在分布式架构中，当系统内发生事件时，系统组件通常需要提供信息给其他组件。发布-订阅模式将关注点分开，以便应用程序可以专注于其核心功能，而消息基础设施则负责处理消息路由和可靠传送等通信职责。发布-订阅模式可实现异步消息发送，从而使发布者和订阅用户解耦。发布者也可以在订阅用户不知情的情况下发送消息。

适用性

在以下情况下使用发布-订阅模式：

- 如果单条消息的工作流程不同，则需要并行处理。
- 无需向多个订阅用户广播消息，也无需接收方的实时响应。
- 系统或应用程序可以容忍数据或状态的最终一致性。
- 应用程序或组件必须与其他可能使用不同语言、协议或平台的应用程序或服务进行通信。

问题和注意事项

- 订阅者可用性：发布者不知道订阅用户是否在收听，订阅用户可能没有收听。发布的消息本质上是短暂的，如果订阅用户不可用，则可能会导致消息被丢弃。
- 消息传送保证：通常，发布-订阅模式不能保证向所有订阅者类型发送消息，尽管某些服务（例如 Amazon Simple Notification Service（Amazon SNS））可以向某些订阅者子集提供精确一次的传送。
- 存活时间（TTL）：消息有生命期，如果未在该时段内处理则过期。考虑将已发布的消息添加到队列中，以便其可以持续存在，并保证在 TTL 期限之后进行处理。

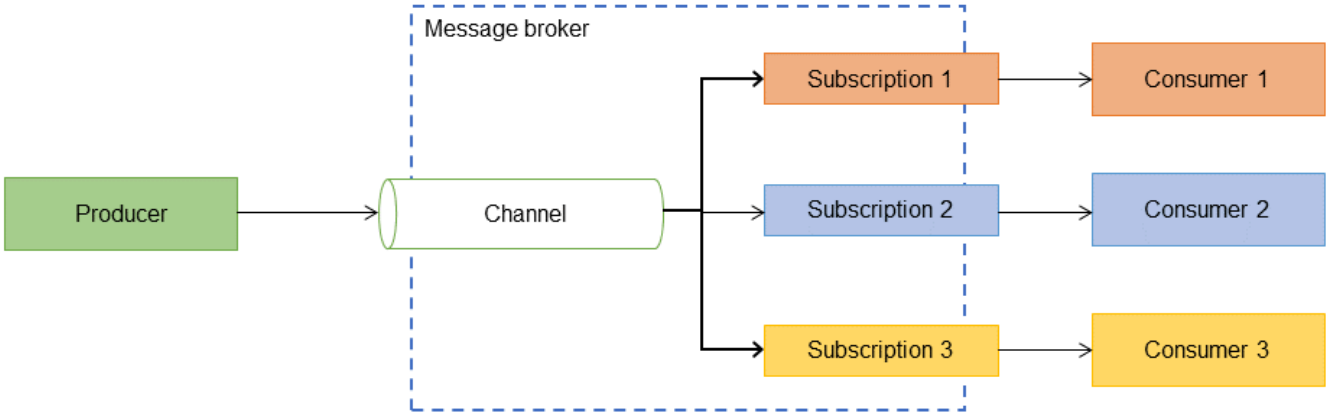
- 消息相关性：生产者可以将相关性的时间跨度设置为消息数据的一部分，并且可以在此日期之后丢弃消息。在决定如何处理消息之前，请考虑设计使用者来检查这些信息。
- 最终一致性：发布消息的时间与订阅用户使用消息的时间之间存在延迟。当需要强一致性时，这样可能会导致订阅用户数据存储最终变得一致。当生产者和使用者需要近乎实时的互动时，最终一致性也可能是一个问题。
- 单向通信：将发布-订阅模式视为单向通信。如果需要同步响应，则需要使用返回订阅通道进行双向消息传递的应用程序应考虑使用请求-回复模式。
- 消息顺序：不能保证消息排序。如果使用者需要有序的消息，则建议您使用 Amazon SNS FIFO 主题来保证顺序。
- 消息复制：基于消息收发基础实施，可以将重复的消息传送给使用者。必须将使用者设计为幂等性，才能处理重复的消息。或者，使用 Amazon SNS FIFO 主题来保证“精确一次”的传送。
- 消息过滤：使用者通常只对生产者发布的消息的子集感兴趣。提供机制，允许订阅用户通过提供主题或内容筛选器来筛选或缩小他们收到的消息范围。
- 消息重播：消息重播功能可能取决于消息收发基础实施。您还可以根据应用场景提供自定义实施。
- 死信队列：在邮政系统中，死信办公室是处理无法投递的邮件的设施。发-订消息收发中，死信队列（DLQ）是指无法传输给订阅端点的消息的队列。

实现

高级架构

在发布-订阅模式中，被称为消息代理或路由器的异步消息收发子系统会跟踪订阅。当生产者发布事件时，消息收发基础设施会向每位使用者发送一条消息。将消息发送给订阅用户后，该消息将从消息基础设施中移除，因此无法重播，新订阅用户也看不到该事件。消息代理或路由器通过以下方式将事件产生器与消息使用者分离：

- 为生产者提供输入通道，以使用定义的消息格式发布打包成消息的事件。
- 为每个订阅创建一个单独的输出通道。订阅是指使用者的连接，他们在其中监听与特定输入通道关联的事件消息。
- 事件发布时，将消息从输入通道复制到所有使用者的输出通道。



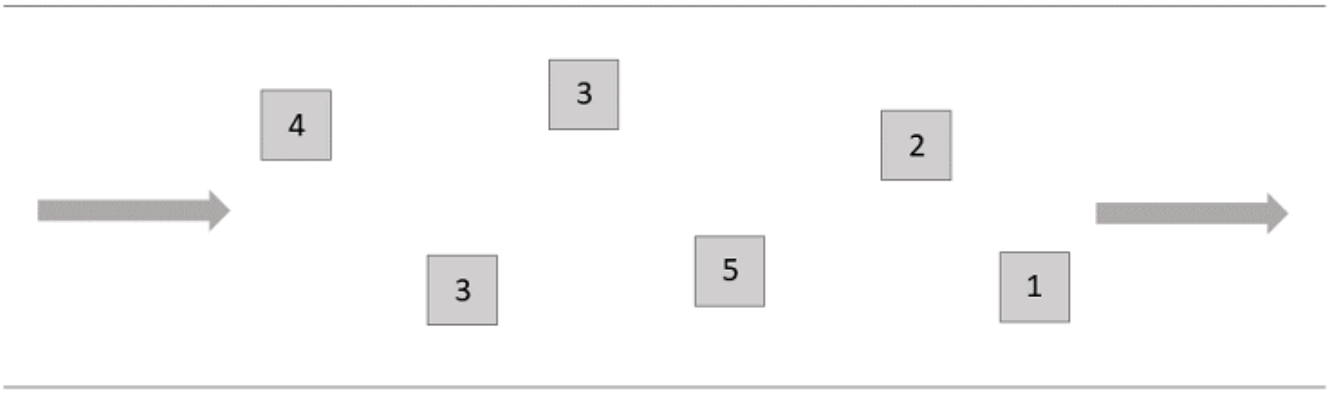
使用亚马逊云科技服务来实施

Amazon SNS

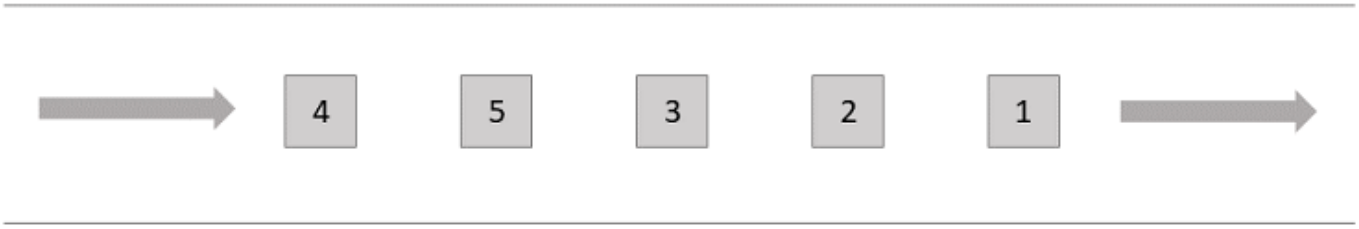
Amazon SNS 是一项完全托管的发布者-订阅用户服务，它提供应用程序对应用程序（A2A）的消息收发，用于分离分布式应用程序。它还提供应用程序对人（A2P）的消息收发，用于发送短信、电子邮件和其他推送通知。

Amazon SNS 提供两种类型的主题：标准主题和先进先出（FIFO）主题。

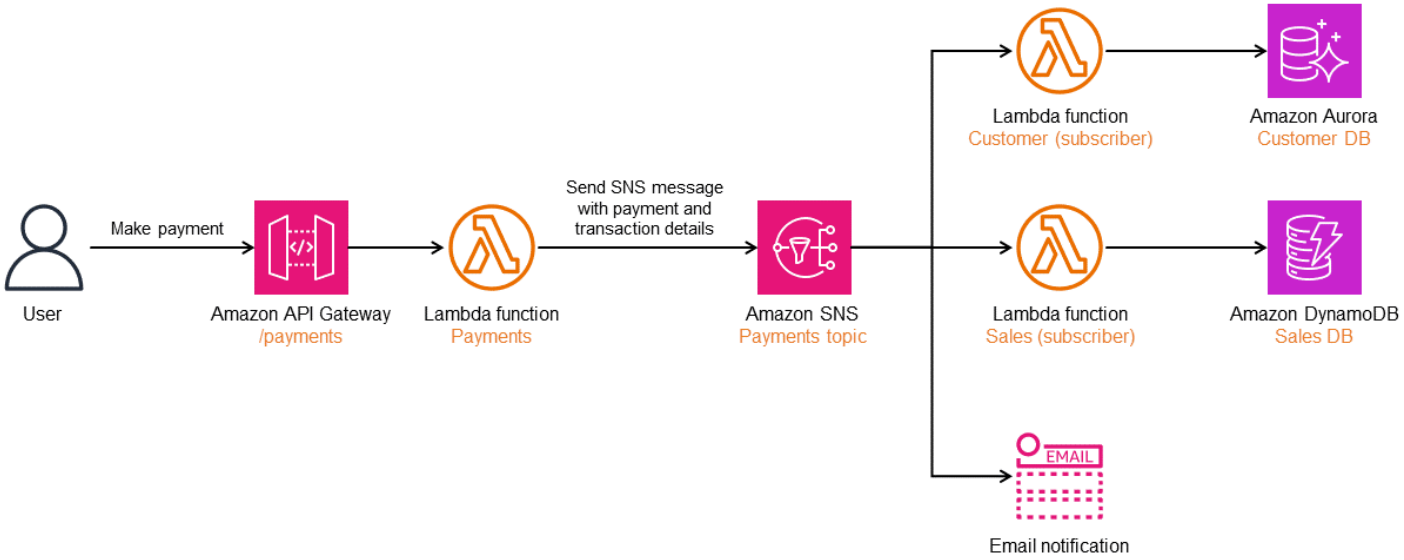
- 标准主题支持每秒无限数量的消息，并提供最佳排序和重复数据删除。



- FIFO 主题提供严格的排序和重复数据删除，每个 FIFO 主题最多支持每秒 300 条消息或每秒 10 MB（按先到者计算）。

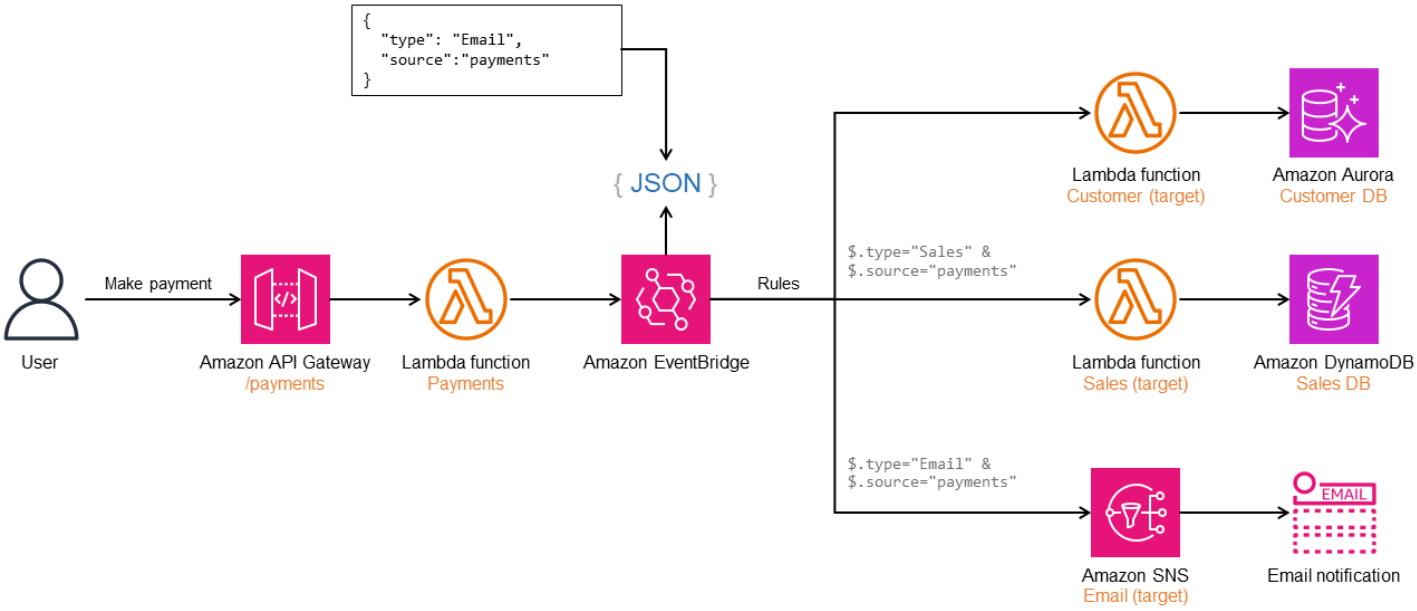


下图显示了如何使用 Amazon SNS 实现发布-订阅模式。用户付款后，Payments Lambda 函数会向 Payments SNS 主题发送一条 SNS 消息。此 SNS 主题有三个订阅用户。每个订阅用户都会收到消息的一个副本并对其进行处理。



Amazon EventBridge

如果您需要更复杂地将来自多个生产者的消息通过不同协议路由到订阅的使用者，或者需要直接订阅和扇出订阅，则可以使用 Amazon EventBridge。EventBridge 还支持基于内容的路由、筛选、排序以及拆分或聚合。在下图中，EventBridge 用于构建发布-订阅模式的版本，在该模式中，使用事件规则定义订阅用户。用户付款后，Payments Lambda 函数使用基于自定义架构的默认事件总线向 EventBridge 发送消息，该架构包含三条指向不同目标的规则。每个微服务都会处理消息并执行所需的操作。



研讨会

- 在亚马逊云科技云上构建事件驱动型架构
- 使用 Amazon Simple Queue Service（Amazon SQS）、Amazon Simple Notification Service（Amazon SNS）发送扇出事件通知

参考博客文章

- 在无服务器应用程序的消息收发服务之间进行选择
- 使用 DLQ 为 Amazon SNS、Amazon SQS、Amazon Lambda 设计耐用的无服务器应用程序
- 使用 Amazon SNS 消息筛选功能简化您的发-订消息收发方式

相关内容

- 发-订消息收发方式的特征

使用退避模式重试

意图

使用退避模式的重试可以透明地重试因暂时错误而失败的操作，从而提高应用程序稳定性。

动机

在分布式架构中，暂时性错误可能是由服务限制、网络连接暂时中断或服务暂时不可用引起的。自动重试因这些暂时错误而失败的操作可改善用户体验和应用程序韧性。但是，频繁重试可能会使网络带宽过载并导致争用。指数回退是一种通过增加指定重试次数的等待时间来重试操作的技术。

适用性

在以下情况下，使用回退重试模式：

- 您的服务经常限制请求以防止过载，从而导致调用过程出现 429 请求过多异常。
- 在分布式架构中，网络是一个看不见的参与者，而暂时的网络问题会导致故障。
- 被调用的服务暂时不可用，导致故障。除非使用此模式引入回退超时，否则频繁重试可能会导致服务降级。

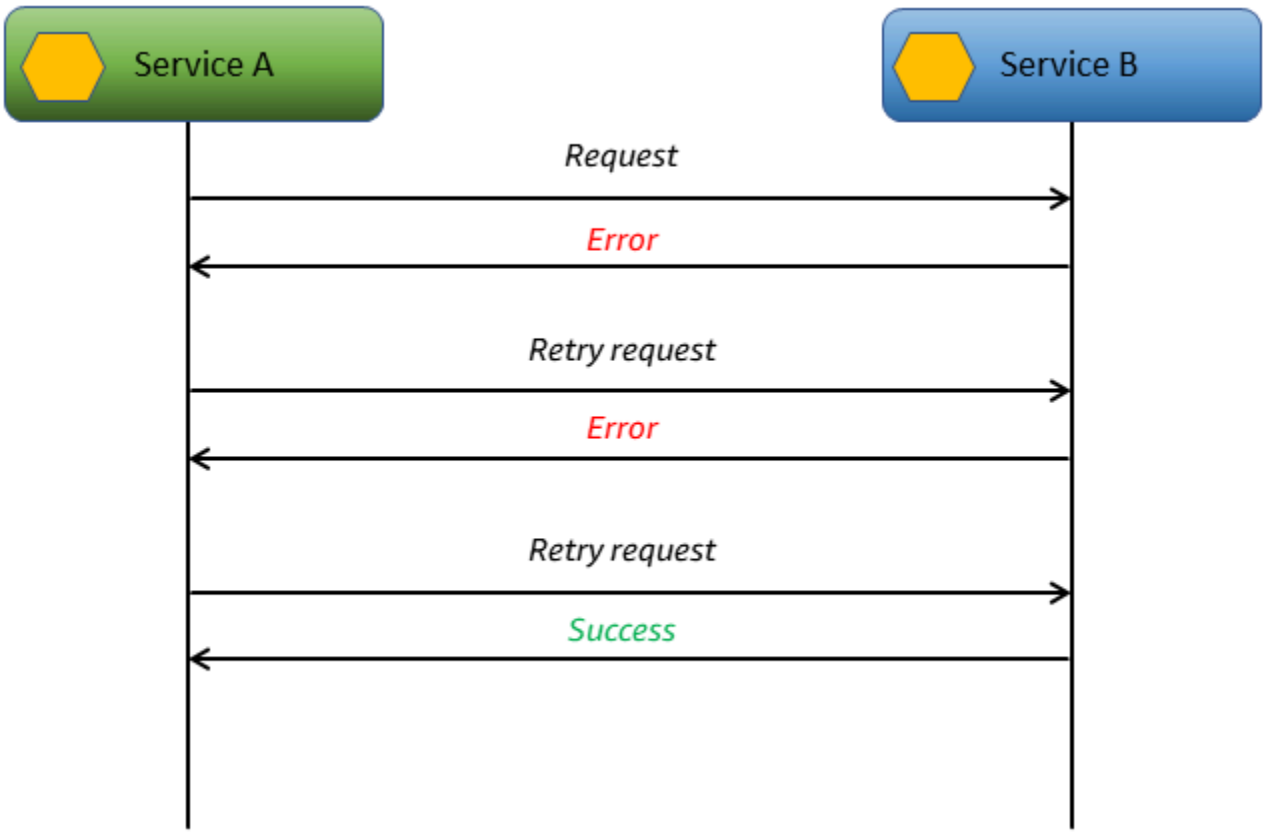
问题和注意事项

- 幂等性：如果对该方法的多次调用与对系统状态的单个调用具有相同的效果，则该操作视为幂等的。使用回退重试模式时，操作应该是幂等操作。否则，部分更新可能会损坏系统状态。
- 网络带宽：如果重试次数过多占用网络带宽，则可能会导致服务降级，从而造成响应时间变慢。
- 快速失效机制场景：对于非瞬态错误，如果可以确定故障原因，则使用断路器模式快速失效会更有效。
- 回退率：引入指数回退可能会影响服务超时，从而导致最终用户的等待时间更长。

实现

高级架构

下图说明了在返回成功响应之前，服务 A 如何重试对服务 B 的调用。如果服务 B 在尝试几次后仍未返回成功的响应，则服务 A 可以停止重试并将失败返回给其调用方。

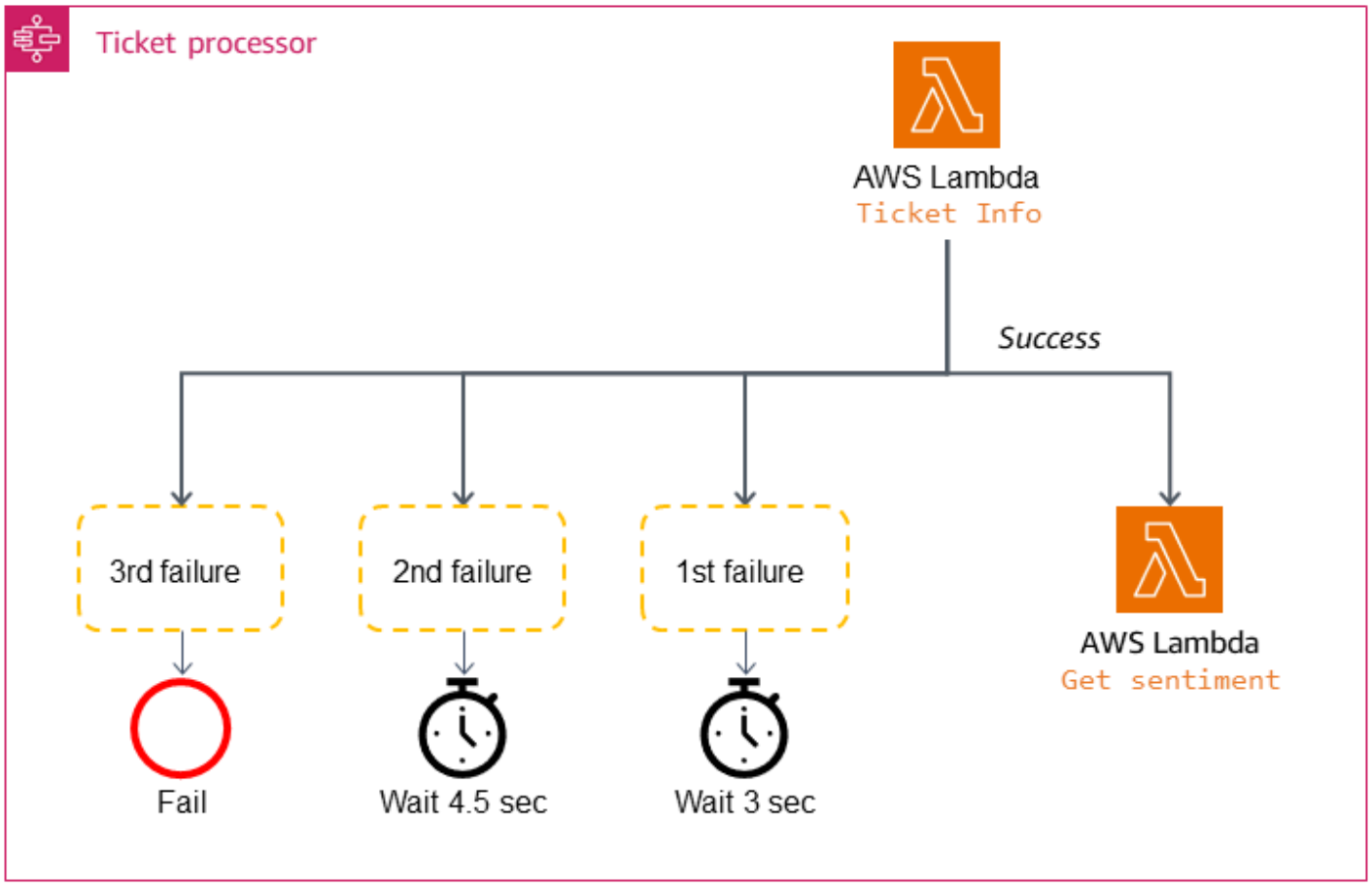


使用 AWS 服务来实施

下图显示了客户支持平台上的工单处理工作流程。通过自动升级工单优先级，可以加快不满意客户的工单。`Ticket info` Lambda 函数提取工单详细信息并调用 `Get sentiment` Lambda 函数。`Get sentiment` Lambda 函数通过将描述传递给 Amazon Comprehend（未显示）来检查客户的情绪。

如果对 `Get sentiment` Lambda 函数的调用失败，则工作流程将重试该操作三次。 AWS Step Functions 允许您配置回退值，从而实现指数回退。

在此示例中，配置了最多重试三次，并设置了 1.5 秒的递增乘数。如果第一次重试发生在 3 秒之后，则第二次重试发生在 3 x 1.5 秒 = 4.5 秒之后，第三次重试发生在 4.5 x 1.5 秒 = 6.75 秒之后。如果第三次重试失败，则工作流程将失败。回退逻辑不需要任何自定义代码 – 它由 AWS Step Functions 作为配置提供。



代码示例

以下代码显示了回退重试模式的实施。

```
public async Task DoRetriesWithBackOff()
  {
     int retries = 0;
     bool retry;
     do
     {
       //Sample object for sending parameters
       var parameterObj = new InputParameter { SimulateTimeout = "false" };
       var content = new StringContent(JsonConvert.SerializeObject(parameterObj),
                                 System.Text.Encoding.UTF8, "application/json");
       var waitInMilliseconds = Convert.ToInt32((Math.Pow(2, retries) - 1) * 100);
       System.Threading.Thread.Sleep(waitInMilliseconds);
       var response =  await _client.PostAsync(_baseURL, content);
       switch (response.StatusCode)
```

```
      {
        //Success
        case HttpStatusCode.OK:
          retry = false;
          Console.WriteLine(response.Content.ReadAsStringAsync().Result);
          break;
        //Throttling, timeouts
        case HttpStatusCode.TooManyRequests:
        case HttpStatusCode.GatewayTimeout:
          retry = true;
          break;
        //Some other error occured, so stop calling the API
        default:
          retry = false;
          break;
      }
      retries++;
    } while (retry && retries < MAX_RETRIES);
  }
```

GitHub 存储库

有关此模式示例架构的完整实施，请参阅 GitHub 存储库 https://github.com/aws-samples/retry-with-backoff。

相关内容

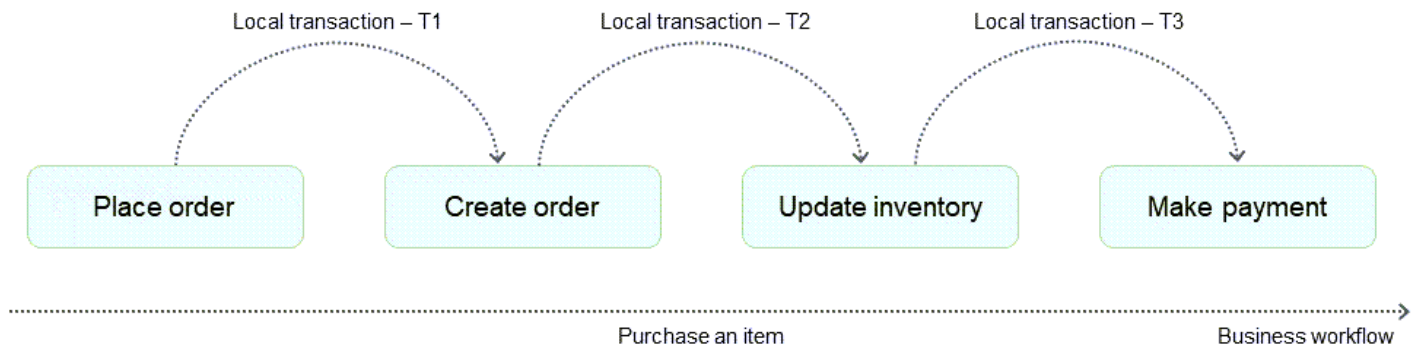
- 超时、重试和抖动回退（Amazon Builders' Library）

Saga 模式

Saga 由一系列本地事务组成。saga 中的每个本地事务都会更新数据库，并触发下一个本地事务。如果事务失败，则 saga 将运行补偿事务，以恢复先前事务所做的数据库更改。

通过使用延续和补偿原则，这种本地事务顺序有助于实现业务工作流程。延续原则决定工作流程的向前恢复，而补偿原则决定向后恢复。如果在事务的任何步骤更新失败，则 saga 会发布一个事件，用于延续（重试事务）或补偿（返回到先前的数据状态）。这样可以确保数据完整性得到维护，并且在数据存储之间保持一致。

例如，当用户从在线零售商处购买图书时，该过程由一系列事务组成，例如订单创建、库存更新、付款和发货，这些事务代表了业务工作流程。为了完成此工作流程，分布式架构会发出一系列本地事务，以便在订单数据库中创建订单、更新库存数据库和更新付款数据库。流程成功后，将依次调用这些事务以完成业务工作流程，如下图所示。但是，如果其中任何一个本地事务失败，则系统应该能够决定适当的下一步 – 即向前恢复或向后恢复。



以下两种场景有助于确定下一步是向前恢复还是向后恢复：

- 平台级故障，即底层基础设施出现问题并导致事务失败。在这种情况下，saga 模式可以通过重试本地事务并延续业务流程来实现向前恢复。
- 应用程序级故障，即由于无效付款而导致付款服务失败。在这种情况下，saga 模式可以通过发出补偿事务来更新库存和订单数据库，并将之恢复为先前的状态来执行向后恢复。

saga 模式处理业务工作流程，并确保通过向前恢复达到理想的最终状态。如果出现故障，它会使用向后恢复来还原本地事务，以避免出现数据一致性问题。

saga 模式有两种变体：编配和编排。

saga 编配

saga 编配模式取决于微服务发布的事件。saga 参与方（微服务）订阅事件并根据事件触发器采取行动。例如，下图中的订单服务会发出一个 `OrderPlaced` 事件。库存服务订阅该事件，并在 `OrderPlaced` 事件发出时更新库存。同样，参与方服务根据所发出事件的上下文行动。

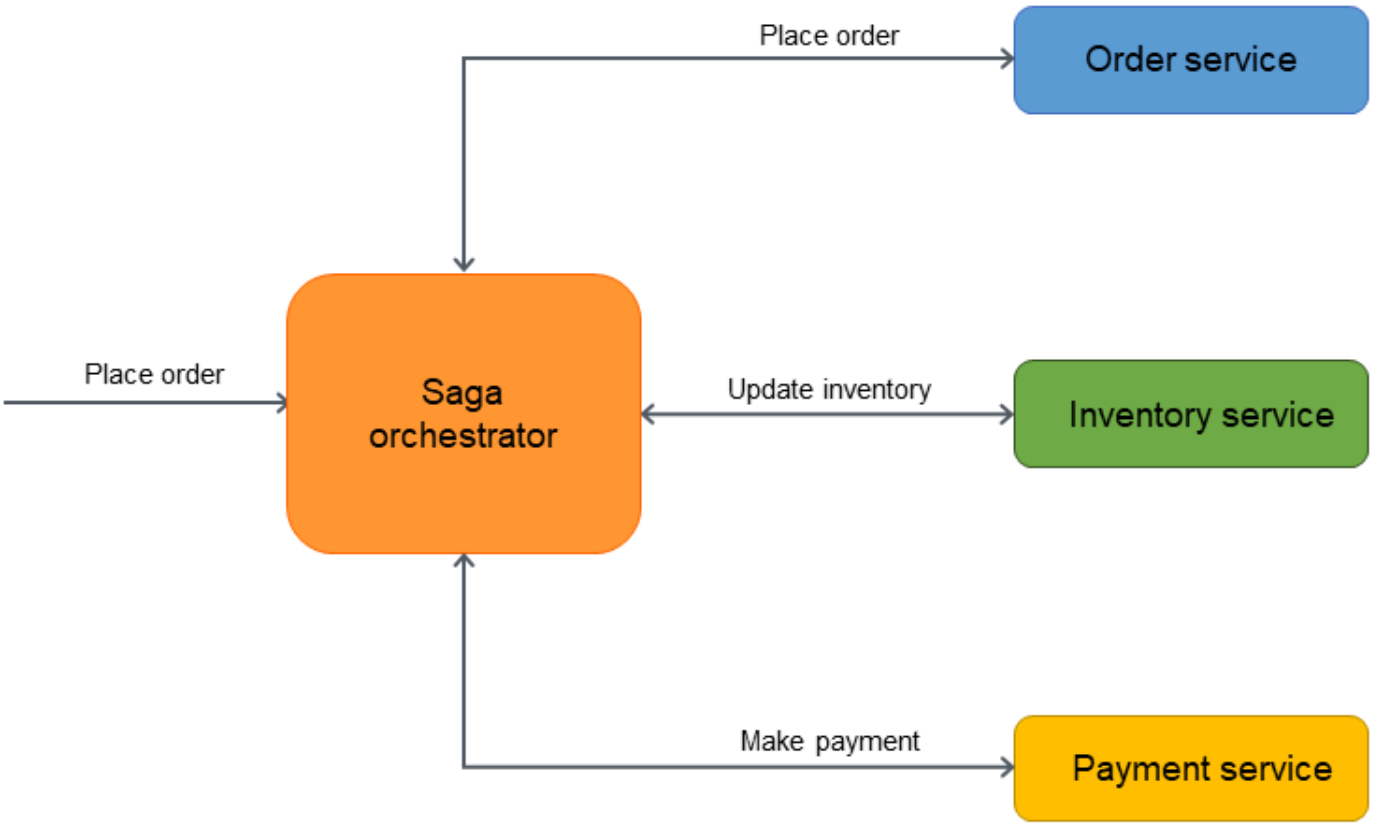
当 saga 中只有几个参与方，并且您需要一个没有单点故障的简单实施时，saga 编配模式是合适的。当添加更多参与方时，使用这种模式跟踪参与方之间的依赖关系就会变得比较困难。

有关详细评述，请参阅本指南的 saga 编配部分。

Saga 编排

saga 编排模式有一个名为编排工具的中央协调器。saga 编排工具管理和协调整个事务生命周期。它知道完成事务需要执行的一系列步骤。要运行某个步骤，它会向参与方微服务发送一条消息以执行该操作。参与方微服务完成操作并向编排工具发送一条消息。根据收到的消息，编排工具决定接下来要在事务中运行哪个微服务。

当参与方众多，并且 saga 参与方之间需要松耦合时，saga 编排模式是合适的。编排工具通过使参与方松耦合来囊括逻辑的复杂性。但是，编排工具可能成为单点故障，因为它控制着整个工作流程。



有关详细评述，请参阅本指南的 saga 编排部分。

Saga 编配模式

意图

通过使用事件订阅，saga 编排模式有助于在跨多个服务的分布式事务中保持数据完整性。在分布式事务中，可以在事务完成之前调用多项服务。当服务将数据存储在不同的数据存储中时，要维护这些数据存储之间的数据一致性可能会很困难。

动机

事务是一个可能涉及多个步骤的单个工作单元，其中要么完全执行所有步骤，要么不执行任何步骤，从而使数据存储保持其一致状态。术语原子性、一致性、隔离和持久性（ACID）定义了事务的属性。关系数据库提供 ACID 事务以维护数据一致性。

为了维护事务的一致性，关系数据库使用两阶段提交（2PC）方法。这包括“准备阶段”和“提交阶段”。

- 在准备阶段，协调过程要求事务的参与进程（参与方）承诺要么提交事务，要么回滚事务。

- 在提交阶段，协调过程会要求参与方提交事务。如果参与方不同意在准备阶段提交，则事务将被回滚。

在遵循 database-per-service设计模式的分布式系统中，两阶段提交不是一种选择。这是因为每个事务分布于不同的数据库中，并且没有单个控制器可以协调类似于关系数据存储中两阶段提交的过程。在这种情况下，一种解决方案是使用 saga 编配模式。

适用性

在以下情况下使用 saga 编配模式：

- 您的系统要求在跨多个数据存储的分布式事务中保持数据完整性和一致性。
- 数据存储（例如，NoSQL 数据库）没有 2PC 提供 ACID 事务，您需要在单个事务中更新多个表，而在应用程序边界内实现 2PC 将是一项复杂的任务。
- 管理参与方事务的中央控制过程可能会成为单点故障。
- saga 的参与方是独立的服务，需要松耦合。
- 业务域中的有界上下文之间存在通信。

问题和注意事项

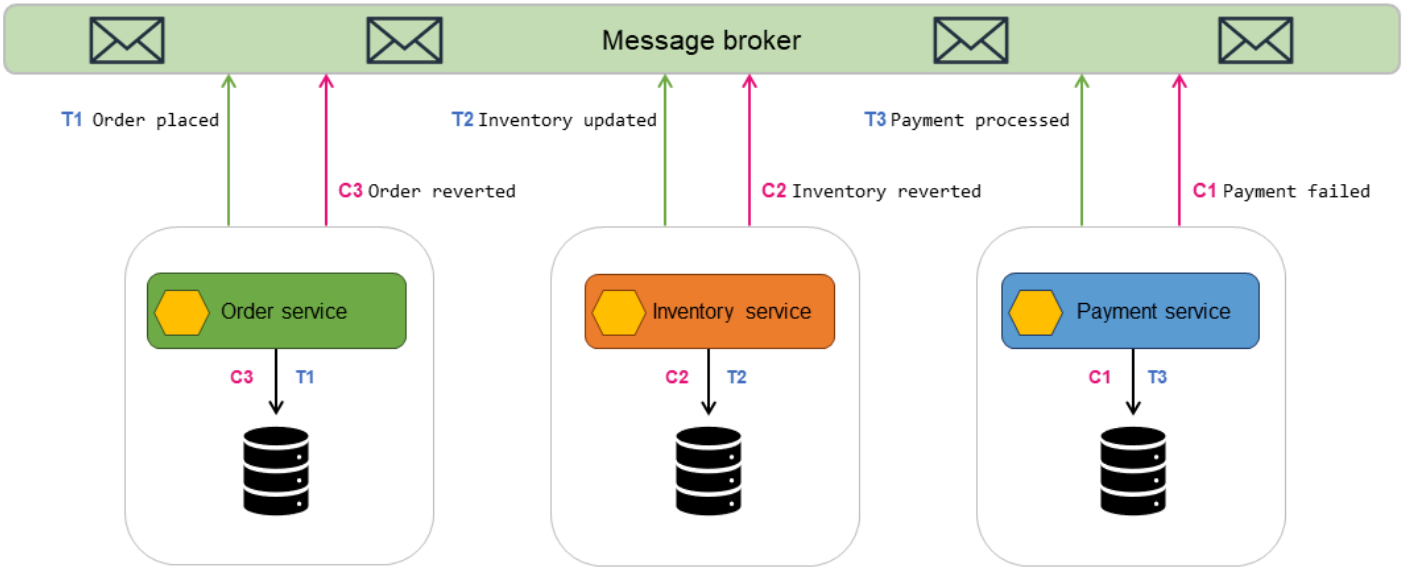
- 复杂性：随着微服务数量的增加，由于微服务之间的交互次数众多，saga 编配可能变得难以管理。复杂性：补偿性事务和重试会增加应用程序代码的复杂性，从而提升维护开销。当 saga 中只有数个参与方，并且您需要一个没有单点故障的简单实施时，编配是合适的。当添加更多参与方时，使用这种模式跟踪参与方之间的依赖关系就会变得比较困难。
- 弹性实施：在 saga 编配中，与 saga 编排相比，在全局范围内实施超时、重试和其他弹性模式更加困难。编配必须在单个组件上实现，而不是在编排工具级别上实现。
- 循环依赖项：参与方使用彼此发布的消息。这样可能会导致循环依赖项，从而让代码变得复杂并带来维护开销，还可能导致死锁。
- 双写问题：微服务必须以原子方式更新数据库和发布事件。任何一个操作失败都可能导致状态不一致。解决这个问题的一种方法是使用事务发件箱模式。
- 保存事件：saga 参与方根据发布的事件行动。出于审计、调试和重播目的，按照事件发生的顺序来保存事件非常重要。您可以使用事件溯源模式将事件保留在事件存储中，以应对为恢复数据一致性需要重播系统状态的情况。事件存储也可用于审计和故障排除目的，因为它们反映了系统中的每一个更改。

- 最终一致性：本地事务的顺序处理可实现最终一致性，这对于需要强一致性的系统来说可能是挑战。您可以通过设定业务团队对一致性模型的期望，或重新评估用例并切换到提供强一致性的数据库来解决此问题。
- 幂等性：Saga 参与方必须具有幂等性，以便在意外崩溃和编排工具故障导致暂时性故障时允许重复执行。
- 事务隔离：saga 模式缺少事务隔离，事务隔离是 ACID 事务中的四个属性之一。事务的隔离程度决定了其他并发事务对该事务所操作的数据的影响程度。事务的并行编排可能会导致数据陈旧。建议使用语义锁定来处理此类场景。
- 可观测性：可观测性是指详细的日志记录和跟踪，以排查实施和编排过程中的问题。当传奇参与者的数量增加导致调试变得复杂时，这一点就变得重要了。 End-to-end与传奇编排相比，在传奇编舞中更难实现监测和报告。
- 延迟问题：当 saga 由几个步骤组成时，补偿性事务可能会增加整体响应时间的延迟。如果事务进行同步调用，则这样会进一步增加延迟。

实施

高级架构

在下面的架构图中，saga 编配有三个参与方：订单服务、库存服务和付款服务。完成事务需要三个步骤：T1、T2 和 T3。三个补偿事务将数据恢复到初始状态：C1、C2 和 C3。



- 订单服务运行本地事务 T1，该事务以原子方式更新数据库并向消息代理发布 `Order placed` 消息。

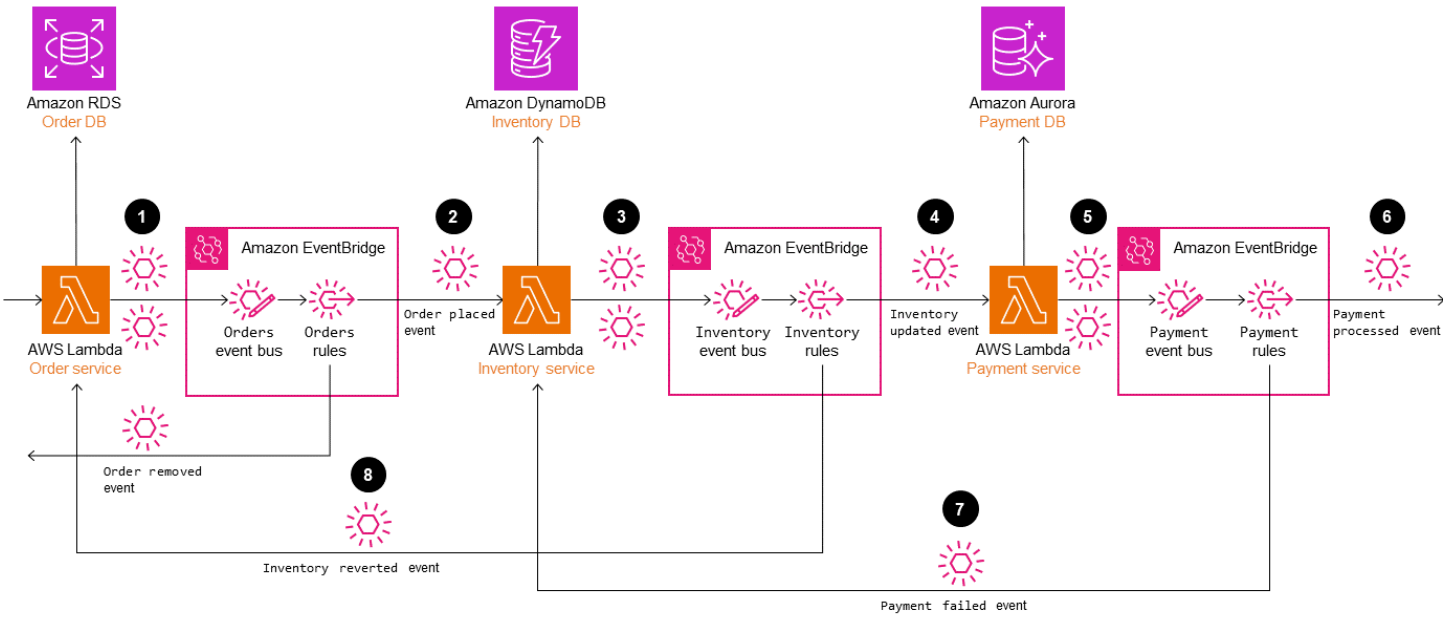
- 库存服务订阅了订单服务消息，并收到已创建订单的消息。
- 库存服务运行本地事务 T2，该事务以原子方式更新数据库并向消息代理发布 `Inventory updated` 消息。
- 付款服务订阅了来自库存服务的消息，并收到已更新库存的消息。
- 付款服务运行本地事务 T3，该事务以原子方式向数据库更新付款详情，并向消息代理发布 `Payment processed` 消息。
- 如果付款失败，支付服务将运行补偿事务 C1，该事务以原子方式恢复数据库中的付款信息，并向消息代理发布 `Payment failed` 消息。
- 运行补偿事务 C2 和 C3 以恢复数据一致性。

使用亚马逊云科技服务来实施

您可以使用 Amazon 实现传奇编舞模式。 EventBridge EventBridge 使用事件来连接应用程序组件。它通过事件总线或管线处理事件。事件总线是接收事件，并将其传送到零个或多个目的地（或目标）的路由器。与事件总线关联的规则会在事件到达时进行评估，并将其发送到目标进行处理。

在以下架构中：

- 微服务（订单服务、库存服务和付款服务）作为 Lambda 函数实施。
- 有三种自定义总 EventBridge 线：`Orders`事件总线、`Inventory`事件总线和`Payment`事件总线。
- `Orders` 规则、`Inventory` 规则和 `Payment` 规则匹配发送到相应事件总线的事件并调用 Lambda 函数。



在成功的场景中，下订单时：

1. 订单服务处理请求并将事件发送到 Orders 事件总线。
2. 这些 Orders 规则与事件相匹配并启动库存服务。
3. 库存服务更新库存并将事件发送到 Inventory 事件总线。
4. Inventory 规则与事件相匹配并启动付款服务。
5. 付款服务处理付款并将事件发送到 Payment 事件总线。
6. Payment 规则匹配事件并将 Payment processed 事件通知发送给侦听器。

   或者，当订单处理出现问题时， EventBridge 规则会启动补偿性交易，以恢复数据更新，以保持数据的一致性和完整性。
7. 如果付款失败，则 Payment 规则会处理该事件并启动库存服务。库存服务部门运行补偿事务以恢复库存。
8. 库存恢复后，库存服务会将 Inventory reverted 事件发送到 Inventory 事件总线。此事件由 Inventory 规则处理。它启动订单服务，该服务运行补偿事务以删除订单。

相关内容

- Saga 编排模式
- 事务发件箱模式

- 使用退避模式重试

Saga 编排模式

意图

Saga 编排模式使用中央协调器（编排工具）来帮助维护跨多个服务的分布式事务中的数据完整性。在分布式事务中，可以在事务完成之前调用多项服务。当服务将数据存储在不同的数据存储中时，要维护这些数据存储之间的数据一致性可能会很困难。

动机

事务是一个可能涉及多个步骤的单个工作单元，其中要么完全执行所有步骤，要么不执行任何步骤，从而使数据存储保持其一致状态。术语原子性、一致性、隔离和持久性（ACID）定义了事务的属性。关系数据库提供 ACID 事务以维护数据一致性。

为了维护事务的一致性，关系数据库使用两阶段提交（2PC）方法。这包括“准备阶段”和“提交阶段”。

- 在准备阶段，协调过程要求事务的参与进程（参与方）承诺要么提交事务，要么回滚事务。
- 在提交阶段，协调过程会要求参与方提交事务。如果参与方不同意在准备阶段提交，则事务将被回滚。

在遵循 database-per-service设计模式的分布式系统中，两阶段提交不是一种选择。这是因为每个事务分布于不同的数据库中，并且没有单个控制器可以协调类似于关系数据存储中两阶段提交的过程。在这种情况下，一种解决方案是使用 saga 编排模式。

适用性

在以下情况下使用 saga 编排模式：

- 您的系统要求在跨多个数据存储的分布式事务中保持数据完整性和一致性。
- 数据存储没有 2PC 来提供 ACID 事务，因此在应用程序边界内实现 2PC 是一项复杂的任务。
- 您有 NoSQL 数据库，而这些数据库不提供 ACID 事务，您需要在单个事务中更新多个表。

问题和注意事项

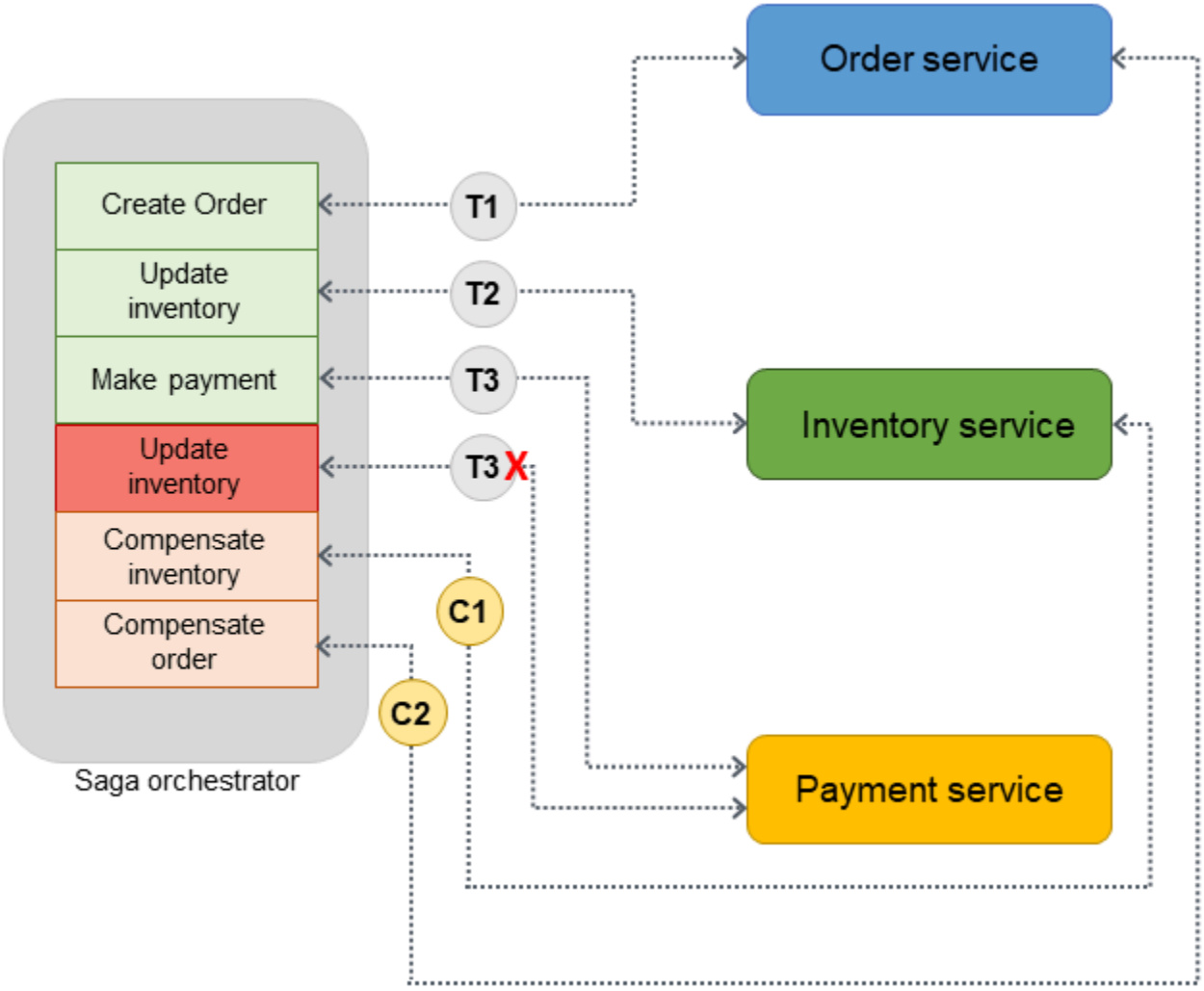
- 复杂性：补偿事务和重试会增加应用程序代码的复杂性，从而带来维护开销。

- 最终一致性：本地事务的顺序处理可实现最终一致性，这对于需要强一致性的系统来说可能是挑战。您可以通过设定业务团队对一致性模型的期望，或通过切换到提供强一致性的数据存储来解决此问题。
- 幂等性：Saga 参与方需要具有幂等性，以便在意外崩溃和编排工具故障导致暂时性故障时允许重复执行。
- 事务隔离：Saga 缺少事务隔离。事务的并行编排可能会导致数据陈旧。建议使用语义锁定来处理此类场景。
- 可观测性：可观测性是指详细的日志记录和跟踪，以排查执行和编排过程中的问题。当 saga 参与方的数量增加导致调试变得复杂时，这一点变得很重要。
- 延迟问题：当 saga 由几个步骤组成时，补偿性事务可能会增加整体响应时间的延迟。在这种情况下，要避免同步调用。
- 单点故障：编排工具可能成为单点故障，因为它协调整个事务。在某些情况下，由于这个问题，首选 saga 编配模式。

实施

高级架构

在下面的架构图中，saga 编排工具有三个参与方：订单服务、库存服务和付款服务。完成事务需要三个步骤：T1、T2 和 T3。saga 编排工具知道这些步骤并按要求的顺序运行它们。当步骤 T3 失败（付款失败）时，编排工具会运行补偿事务 C1 和 C2，将数据恢复到初始状态。



当事务分布在多个数据库中时，您可以使用 AWS Step Functions 来实现 saga 编排。

使用 AWS 服务实施

示例解决方案使用 Step Functions 中的标准工作流程来实现 saga 编排模式。

当客户调用 API 时，也会调用 Lambda 函数，并在 Lambda 函数中进行预处理。该函数启动了 Step Functions 工作流程，以开始处理分布式事务。如果不需要预处理，则无需使用 Lambda 函数即可直接从 API Gateway 启动 Step Functions 工作流程。

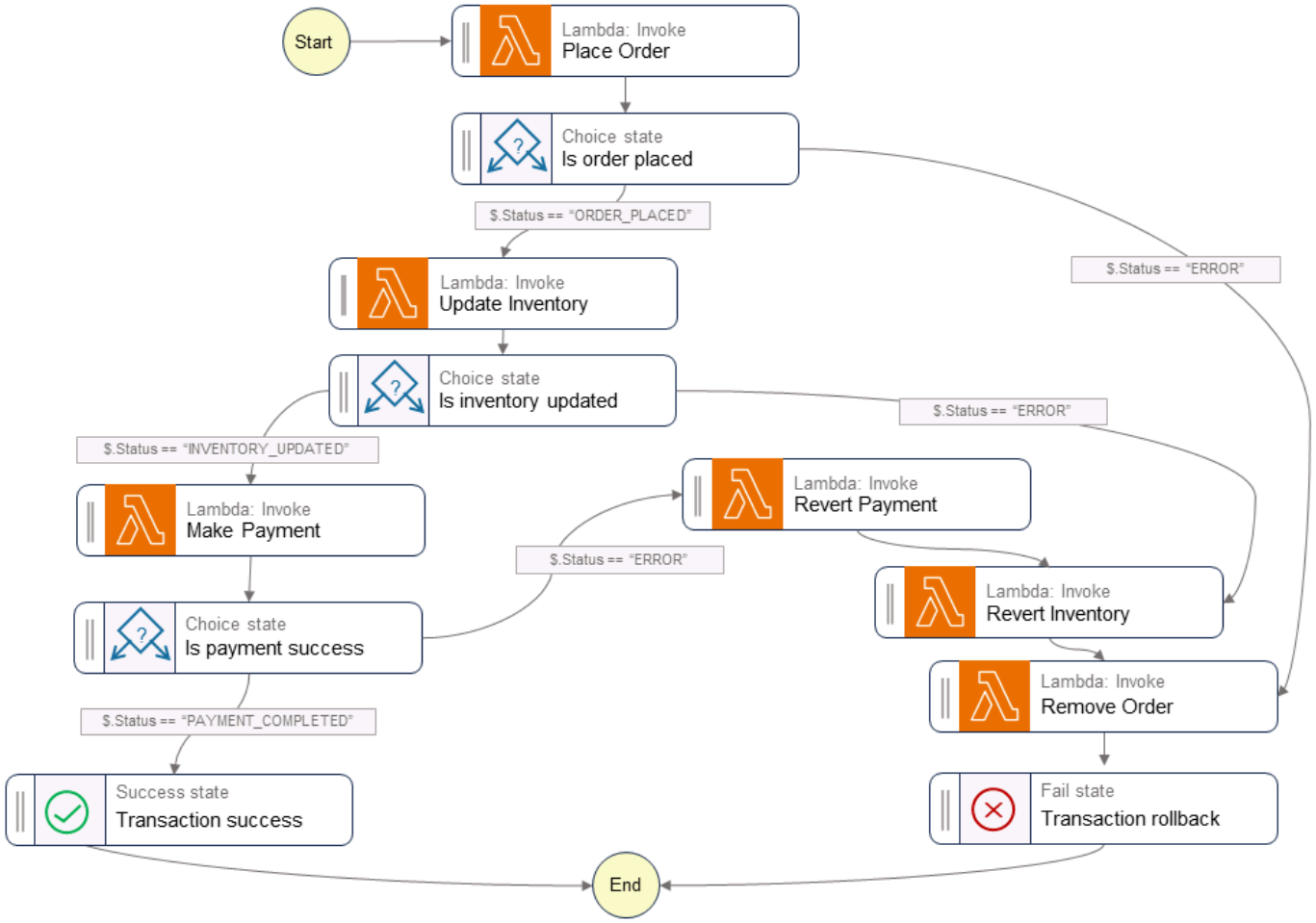
使用 Step Functions 可以缓解单点故障问题，这是 saga 编排模式的实施所固有的。Step Functions 具有内置容错特性，可在每个亚马逊云科技区域的多个可用区中维护服务容量，以保护应用程序免受单个计算机或数据中心故障的影响。这有助于同时确保服务本身及其运行的应用程序工作流程的高可用性。

Saga Step Functions 工作流程

Step Functions 状态机允许您为模式实施配置基于决策的控制流要求。Step Functions 工作流程调用用于下单、库存更新和付款处理的各个服务以完成事务，并发送事件通知以便进一步处理。Step Functions 工作流程充当编排工具来协调事务。如果工作流程包含任何错误，则编排工具会运行补偿性事务，以跨服务维护数据完整性。

下图显示了 Step Functions 工作流程中运行的步骤。`Place Order`、`Update Inventory` 和 `Make Payment` 步骤指示了成功之路。下订单、更新库存并处理付款，然后再将 `Success` 状态返回给调用方。

`Revert Payment`、`Revert Inventory` 和 `Remove Order` Lambda 函数表示当工作流程中的任何步骤失败时，编排工具运行的补偿性事务。如果工作流程在 `Update Inventory` 步骤失败，则编排工具会在向调用方返回 `Fail` 状态之前调用 `Revert Inventory` 和 `Remove Order` 步骤。这些补偿事务确保数据完整性得到维护。库存恢复到其原始水平，将订单恢复。



代码示例

以下示例代码显示了如何使用 Step Functions 创建 saga 编排工具。要查看完整的代码，请参阅此示例的GitHub存储库。

任务定义

```
var successState = new Succeed(this,"SuccessState");
var failState = new Fail(this, "Fail");

var placeOrderTask = new LambdaInvoke(this, "Place Order", new LambdaInvokeProps
{
    LambdaFunction = placeOrderLambda,
    Comment = "Place Order",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});
```

```
var updateInventoryTask = new LambdaInvoke(this,"Update Inventory", new
 LambdaInvokeProps
{
    LambdaFunction = updateInventoryLambda,
    Comment = "Update inventory",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});

var makePaymentTask = new LambdaInvoke(this,"Make Payment", new LambdaInvokeProps
{
    LambdaFunction = makePaymentLambda,
    Comment = "Make Payment",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
});

var removeOrderTask = new LambdaInvoke(this, "Remove Order", new LambdaInvokeProps
{
    LambdaFunction = removeOrderLambda,
    Comment = "Remove Order",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(failState);

var revertInventoryTask = new LambdaInvoke(this,"Revert Inventory", new
 LambdaInvokeProps
{
    LambdaFunction = revertInventoryLambda,
    Comment = "Revert inventory",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(removeOrderTask);

var revertPaymentTask = new LambdaInvoke(this,"Revert Payment", new LambdaInvokeProps
{
    LambdaFunction = revertPaymentLambda,
    Comment = "Revert Payment",
    RetryOnServiceExceptions = false,
    PayloadResponseOnly = true
}).Next(revertInventoryTask);

var waitState = new Wait(this, "Wait state", new WaitProps
```

```
{
    Time = WaitTime.Duration(Duration.Seconds(30))
}).Next(revertInventoryTask);
```

Step function 和状态机定义

```
var stepDefinition = placeOrderTask
                .Next(new Choice(this, "Is order placed")
                    .When(Condition.StringEquals("$.Status", "ORDER_PLACED"),
 updateInventoryTask
                        .Next(new Choice(this, "Is inventory updated")
                            .When(Condition.StringEquals("$.Status",
 "INVENTORY_UPDATED"),
                                makePaymentTask.Next(new Choice(this, "Is payment
 success")
                                    .When(Condition.StringEquals("$.Status",
 "PAYMENT_COMPLETED"), successState)
                                    .When(Condition.StringEquals("$.Status", "ERROR"),
 revertPaymentTask)))
                            .When(Condition.StringEquals("$.Status", "ERROR"),
 waitState)))
                    .When(Condition.StringEquals("$.Status", "ERROR"), failState));

var stateMachine = new StateMachine(this, "DistributedTransactionOrchestrator", new
 StateMachineProps {
    StateMachineName = "DistributedTransactionOrchestrator",
    StateMachineType = StateMachineType.STANDARD,
    Role = iamStepFunctionRole,
    TracingEnabled = true,
    Definition = stepDefinition
});
```

GitHub 存储库

有关此模式示例架构的完整实现，请参见 GitHub 存储库，网址为https://github.com/aws-samples/saga-orchestration-netcore-blog。

参考博客文章

- 使用 Saga 编排模式构建无服务器分布式应用程序

相关内容

- Saga 编配模式
- 事务发件箱模式

视频

以下视频讨论了如何使用 AWS Step Functions实现 saga 编排模式。

分散-收集模式

意图

分散-收集模式是一种消息路由模式，它涉及向多个接收者广播相似或相关的请求，并使用称为聚合器的组件将它们的响应聚合回单条消息。此模式有助于实现并行化、减少处理延迟和处理异步通信。使用同步方法可以轻松实现分散-收集模式，但更强大的方法是将其作为异步通信中的消息路由来实现，无论是否使用消息收发服务。

动机

在应用程序处理中，一个可能需要很长时间才能按顺序处理的请求可以分为多个并行处理的请求。您还可以通过 API 调用向多个外部系统发送请求以获得响应。当您需要来自多个来源的输入时，分散-收集模式非常有用。分散-收集聚合结果，以帮助您做出明智的决策或为请求选择最佳响应。

顾名思义，分散-收集模式由两个阶段组成：

- 分散阶段处理请求消息，并将其并行发送给多个接收者。在此阶段，应用程序将请求分散到网络中，并继续运行，无需等待即时响应。
- 在收集阶段，应用程序收集接收者的响应，并对其进行筛选或合并，形成统一的响应。收集完所有响应后，可以将它们聚合为单个响应，也可以选择最佳响应进行进一步处理。

适用性

在以下情况下使用分散-收集模式：

- 您计划汇总和合并各种数据 APIs 以创建准确的响应。该模式将来自不同来源的信息整合成一个统一的整体。例如，预订系统可以向多个接收者发出请求，以获取来自多个外部合作伙伴的报价。
- 必须将同一请求同时发送给多个接收者才能完成事务。例如，您可以使用此模式并行查询库存数据，以检查产品的可用性。
- 您希望实现一个可靠且可扩展的系统，通过将请求分发给多个接收者来实现负载均衡。如果一个接收者失败或负载过高，其他接收者仍然可以处理请求。
- 在实现涉及多个数据来源的复杂查询时，您需要优化性能。您可以将查询分散到相关数据库，收集部分结果，然后将它们组合成一个全面的答案。

- 您正在实现一种 map-reduce 处理，其中数据请求路由到多个数据处理端点进行分片和复制。对部分结果进行筛选和组合，以构成正确的响应。
- 您希望在键值数据库中写入密集型工作负载中跨分区键空间分配写入操作。聚合器通过查询每个分片中的数据来读取结果，然后将它们整合成单个响应。

问题和注意事项

- 容错能力：此模式依赖多个并行工作的接收者，因此妥善处理故障至关重要。为了减轻接收者故障对整个系统的影响，您可以实施冗余、复制和故障检测等策略。
- 横向扩展限制：随着处理节点总数的增加，关联的网络开销也会增加。每个涉及网络通信的请求都会增加延迟，且对并行化的好处产生负面影响。
- 响应时间瓶颈：对于需要在最终处理完成之前处理所有接收者的操作，整个系统的性能会受到最慢接收者响应时间的限制。
- 部分响应：在请求分散到多个接收者时，某些接收者可能会超时。在这些情况下，实现应告知客户端该响应是不完整的。您还可以使用界面前端显示响应聚合详细信息。
- 数据一致性：在您处理跨多个接收者的数据时，必须仔细考虑数据同步和冲突解决技术，以确保最终聚合结果的准确性和一致性。

实施

高级架构

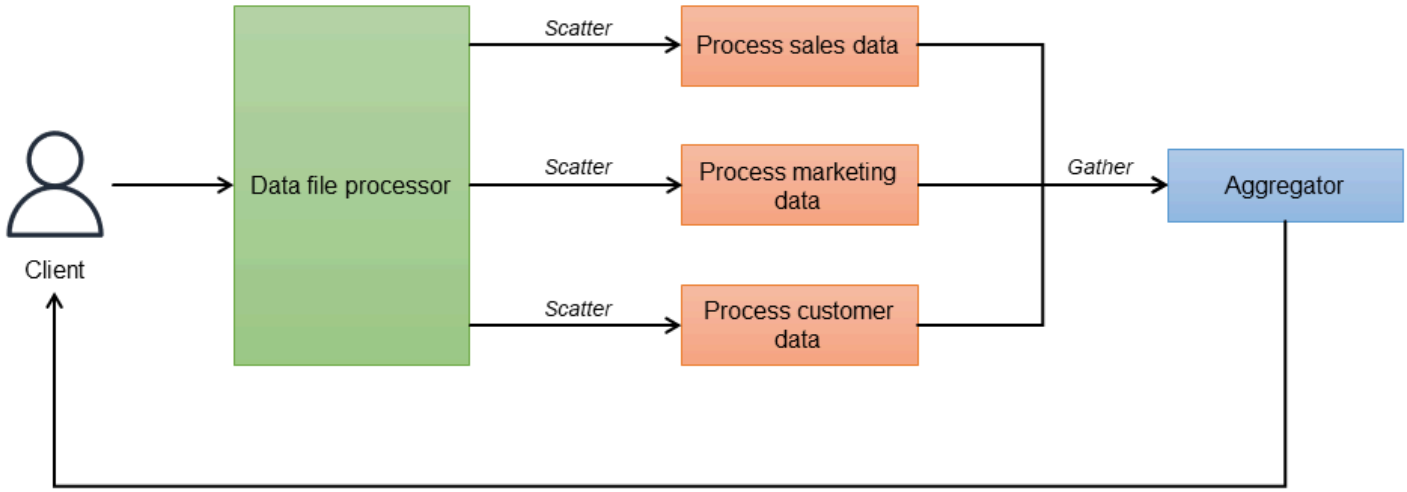
分散-收集模式使用根控制器将请求分发给将处理请求的接收者。在分散阶段，此模式可以使用两种机制向接收者发送消息：

- 按分发分散：应用程序有一个已知的接收者列表，必须调用这些接收者才能获得结果。接收者可以是具有独特功能的不同流程，也可以是已横向扩展以分发处理负载的单个流程。如果任何处理节点超时或显示响应延迟，则控制器可以将处理重新分发给另一个节点。
- 按拍卖分散：应用程序使用发布/订阅模式向感兴趣的接收者广播消息。在这种情况下，接收者可以随时订阅消息或取消订阅。

按分发分散

在按分发分散方法中，根控制器将传入的请求分成独立的任务，并将它们分配给可用的接收者（分散阶段）。每个接收者（进程、容器或 Lambda 函数）独立并行地进行计算，并生成部分响应。在接收者

完成任务时，他们会将响应发送给聚合器（收集阶段）。聚合器合并部分响应并将最终结果返回给客户端。下图说明了此工作流程。

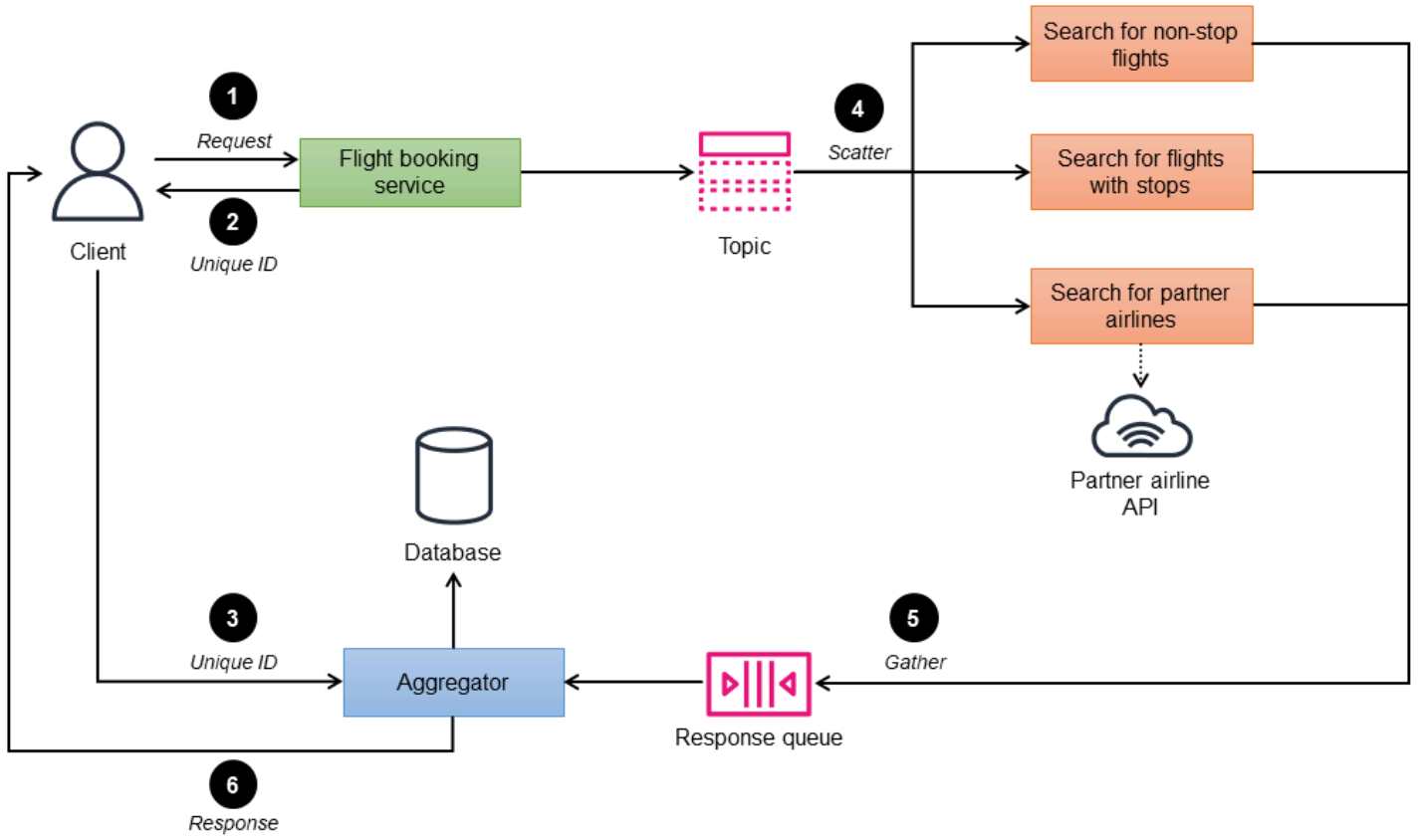


控制器（数据文件处理器）负责编排整组调用，并识别要调用的所有预订端点。它可以配置超时参数，以忽略花费时间过长的响应。发送请求后，聚合器会等待来自每个端点的响应。为了实现韧性，可以将每个微服务与多个实例一起部署，以实现负载均衡。聚合器获取结果，将它们组合成单条响应消息，并在进一步处理之前删除重复数据。超时的响应被忽略。控制器也可以充当聚合器，而不是使用单独的聚合器服务。

按拍卖分散

如果控制器不知道接收者，或者接收者是松耦合的，您可以使用按拍卖分散方法。在此方法中，接收者订阅主题，控制器向该主题发布请求。接收者将结果发布到响应队列。由于根控制器不知道接收者，因此收集过程使用聚合器（另一种消息收发模式）来收集响应，并将其提炼成单个响应消息。聚合器使用唯一 ID 来识别一组请求。

例如，在下图中，按拍卖分散方法用于实现航空公司网站的航班预订服务。该网站允许用户搜索和显示航空公司自身及其合作伙伴航空公司的航班，且必须实时显示搜索状态。航班预订服务由三个搜索微服务组成：直飞航班、中转航班和合作伙伴航空公司。合作伙伴航空公司搜索会调用合作伙伴的 API 端点来获取响应。

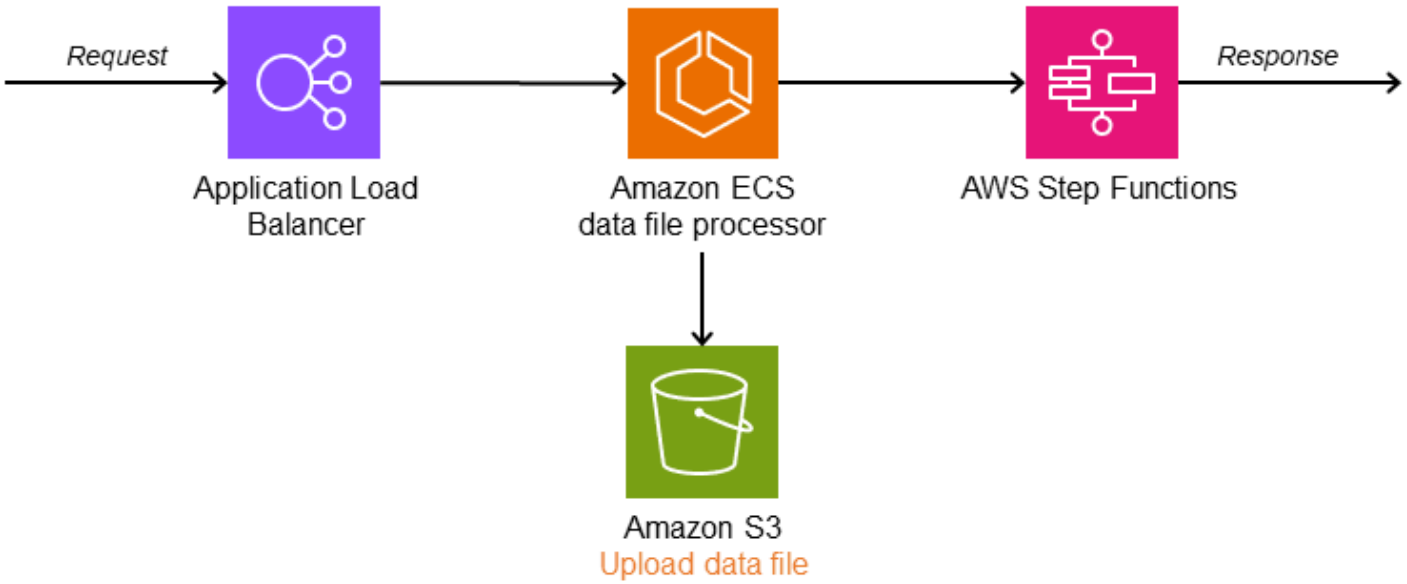


1. 航班预订服务（控制器）将搜索条件作为客户端的输入，然后处理请求并将其发布到该主题。
2. 控制器使用唯一 ID 来识别一组请求。
3. 客户端将唯一 ID 发送给步骤 6 的聚合器。
4. 已订阅预订主题的预订搜索微服务会收到请求。
5. 微服务处理请求，并将给定搜索条件的席位可用性返回到响应队列。
6. 聚合器整理存储在临时数据库中的所有响应消息，按唯一 ID 对航班进行分组，创建单个统一响应，然后将其发送回客户端。

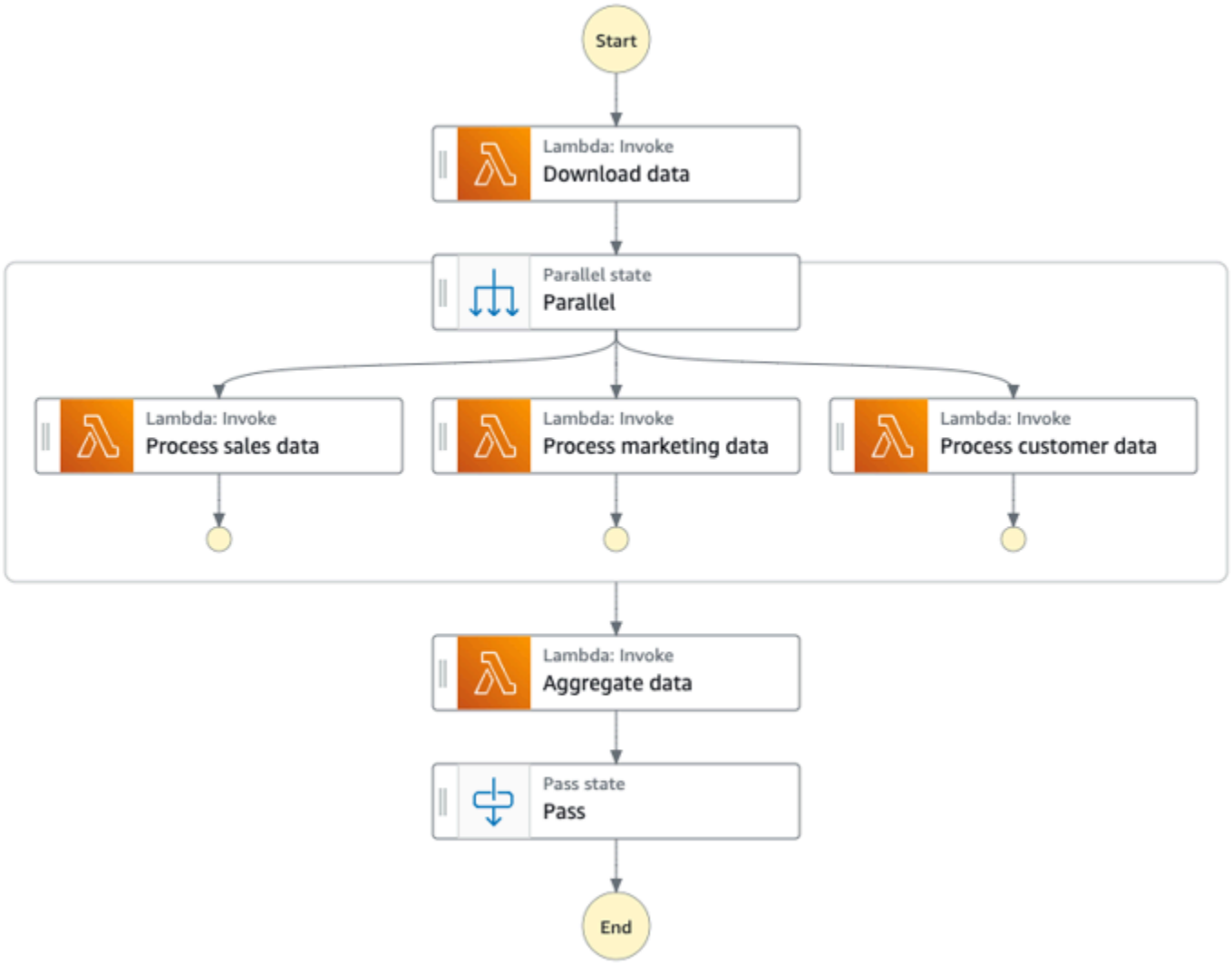
使用实现 AWS 服务

按分发分散

在以下架构中，根控制器是一个数据文件处理器 (Amazon ECS)，它将传入的请求数据拆分为单独的亚马逊简单存储服务 (Amazon S3) 存储桶并启动工作流程。 AWS Step Functions 该工作流程下载数据，并启动并行文件处理。`Parallel` 状态等待所有任务返回响应。 AWS Lambda 函数会聚合数据并将其保存回 Amazon S3。

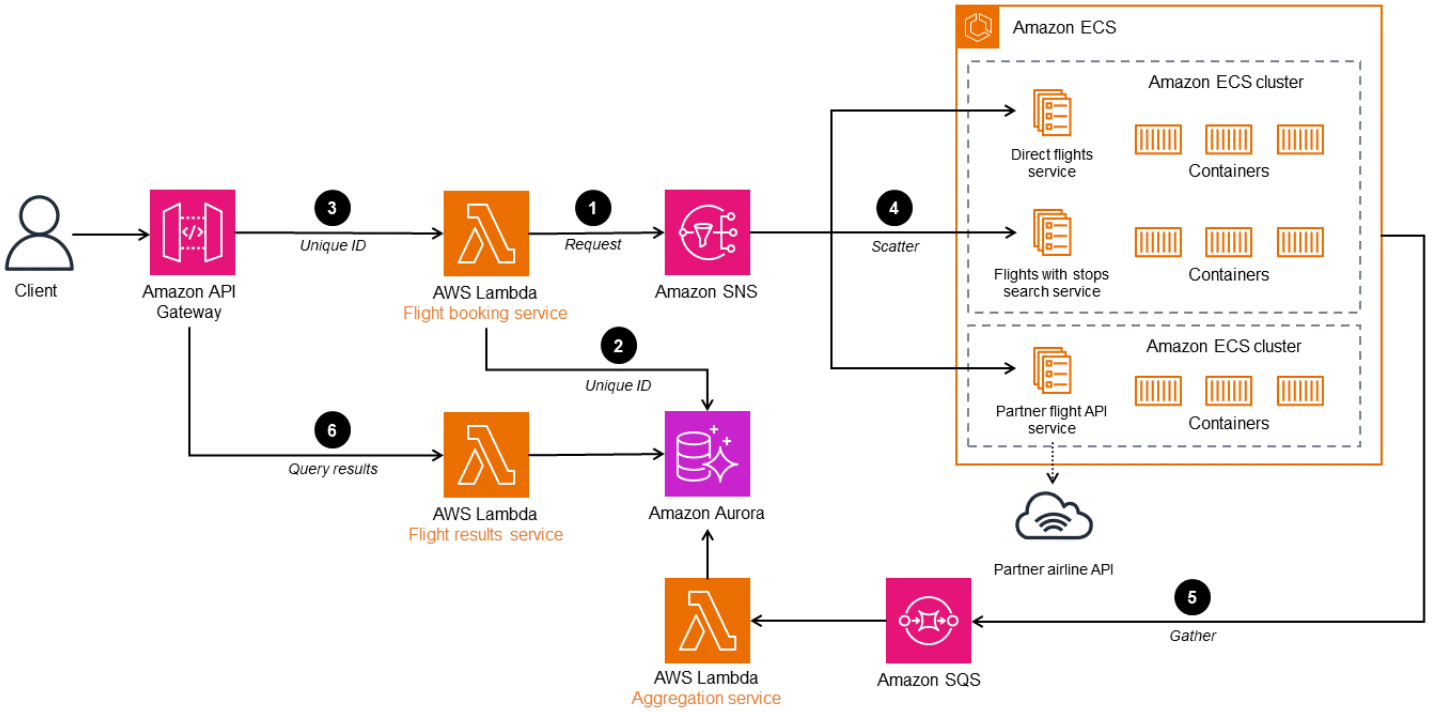


下图说明了处于 `Parallel` 状态的 Step Functions 工作流程。



按拍卖分散

下图显示了按拍卖方法分散的 AWS 架构。根控制器航班预订服务将航班搜索请求分散到多个微服务。发布/订阅渠道是通过 Amazon Simple Notification Service（Amazon SNS）实现的，它是一种用于通信的托管式消息收发服务。Amazon SNS 支持解耦的微服务应用程序之间的消息或与用户的直接通信。您可以在 Amazon Elastic Kubernetes Service（Amazon EKS）或 Amazon Elastic Container Service（Amazon ECS）上部署收件者微服务，以实现更好的管理和可扩展性。航班结果服务将结果返回到客户端。它可以在其他容器编排服务（例如 Amazon ECS AWS Lambda 或 Amazon EKS）中实现。



1. 航班预订服务（控制器）将搜索条件作为客户端的输入，然后处理请求并将其发布到 SNS 主题。
2. 控制器将唯一 ID 发布到 Amazon Aurora 数据库以识别请求。
3. 客户端将唯一 ID 发送给步骤 6 的客户端。
4. 已订阅预订主题的预订搜索微服务会收到请求。
5. 微服务处理请求，并将给定搜索条件的席位可用性返回到 Amazon Simple Queue Service（Amazon SQS）中的响应队列。聚合器整理所有响应消息，并将其存储在临时数据库中。
6. 航班结果服务按唯一 ID 对航班进行分组，创建单个统一响应，然后将其发送回客户端。

如果要向此架构添加其他航空公司搜索，您可以添加微服务以订阅 SNS 主题并发布到 SQS 队列。

总而言之，分散-收集模式使分布式系统能够实现高效的并行化、减少延迟并无缝处理异步通信。

GitHub 存储库

有关此模式示例架构的完整实现，请参阅 https://github.com/aws-samples/asynchronous-messaging-workshop/tree/master/code/lab-3 中的 GitHub 存储库。

研讨会

- 解耦微服务讲习会中的分散-收集实验室

参考博客文章

- Application integration patterns for microservices

相关内容

- 发布/订阅模式

strangler fig 模式

意图

strangler fig 模式有助于逐步将单体应用程序迁移到微服务架构，从而降低转型风险和业务中断。

动机

开发单体应用程序是为了在单个流程或容器中提供大部分功能。代码紧密耦合。因此，应用程序更改需要进行彻底的重新测试，以避免出现回归问题。这些更改无法单独进行测试，这会影响周期时间。随着应用程序的功能越来越丰富，高度复杂性会导致更多的维护时间，增加上市时间，从而减缓产品创新。

当应用程序规模扩大时，它会增加团队的认知负担，并可能导致团队所有权界限不明确。根据负载扩展单个功能是不可能的，必须扩展整个应用程序以支持峰值负载。随着系统的老化，该技术可能会过时，从而推高支持成本。单体遗留应用程序遵循开发时可用的最佳实践，并非为分发而设计。

当单体应用程序迁移到微服务架构时，可以将其拆分为较小的组件。这些组件可以独立扩展，可以独立发布，也可以由各个团队拥有。这会导致更快的更改速度，因为更改是局部的，可以快速测试和发布。更改的影响范围较小，因为组件是松耦合的，可以单独部署。

通过重写或重构代码将单体完全替换为微服务应用程序是一项艰巨的任务，也是一个很大的风险。大爆炸式迁移即在单个操作中迁移单体，这会带来转型风险和业务中断。在重构应用程序时，添加新功能极其困难，甚至是不可能的。

解决此问题的一种方法是使用 Martin Fowler 引入的 strangler fig 模式。此模式涉及通过逐步提取功能并围绕现有系统创建新应用程序，来迁移到微服务。单体中的功能逐渐被微服务所取代，应用程序用户可以逐步使用新迁移的功能。当所有功能都迁移到新系统时，可以安全地停用单体应用程序。

适用性

在以下情况下使用 strangler fig 模式：

- 您想逐步将单体应用程序迁移到微服务架构。
- 由于单体规模庞大且复杂，采用大爆炸式迁移方法存在风险。
- 该企业想要添加新功能，迫不及待地想完成转型。
- 在转型期间，必须将最终用户受到的影响降至最低。

问题和注意事项

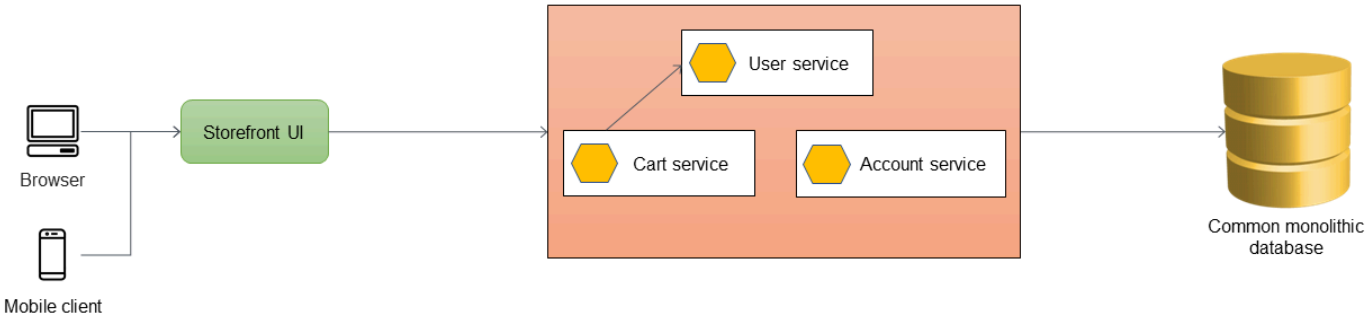
- 代码库访问权限：要实现 strangler fig 模式，您必须有权访问单体应用程序的代码库。随着功能从单体中迁移，您将需要对代码进行细微的更改，并在单体内实现防腐层，以将调用路由到新的微服务。如果没有代码库访问权限，则无法拦截调用。代码库访问权限对于重定向传入请求也至关重要，可能需要进行一些代码重构，以便代理层可以拦截对迁移功能的调用并将其路由到微服务。
- 领域不明确：过早地对系统进行分解可能会付出高昂的代价，尤其是在领域不明确的情况下，并且可能会错误地划分服务边界。领域驱动型设计（DDD）是一种理解领域的机制，而事件风暴是一种确定领域边界的技术。
- 识别微服务：您可以使用 DDD 作为识别微服务的关键工具。要识别微服务，请寻找服务类别之间的自然划分。许多服务将拥有自己的数据访问对象，且可以轻松解耦。具有相关业务逻辑的服务和没有依赖关系或几乎没有依赖关系的类别是微服务的理想选择。您可以在分解单体之前重构代码，以防止紧密耦合。您还应该考虑合规性要求、发布节奏、团队的地理位置、扩展需求、使用案例驱动型技术需求以及团队的认知负荷。
- 防腐层：在迁移过程中，当单体内的功能必须调用作为微服务迁移的功能时，您应该实现防腐层（ACL），将每次调用路由到相应的微服务。为解耦单体内的现有调用方并防止对其进行更改，ACL 可用作适配器或 Facade，以将调用转换为较新的接口。本指南前面的 ACL 模式的“实现”部分对此进行了详细讨论。
- 代理层故障：在迁移期间，代理层会拦截发送到单体应用程序的请求，并将这些请求路由到遗留系统或新系统。但是，此代理层可能成为单点故障或性能瓶颈。
- 应用程序复杂性：大型单体从 strangler fig 模式中获益最大。对于小型应用程序而言，由于完全重构的复杂性较低，与其迁移应用程序，不如采用微服务架构重写应用程序，这样可能更有效率。
- 服务交互：微服务可以同步或异步通信。需要同步通信时，考虑超时是否会导致连接或线程池消耗，从而导致应用程序性能问题。在这种情况下，对于可能长时间失败的操作，使用断路器模式可立即返回失败结果。异步通信可以通过使用事件和消息队列来实现。
- 数据聚合：在微服务架构中，数据分布在数据库之间。需要数据聚合时，您可以在前端使用 AWS AppSync，或者在后端使用命令查询责任分割（CQRS）模式。
- 数据一致性：微服务拥有自己的数据存储，单体应用程序也可能使用此数据。要启用共享，您可以使用队列和代理将新微服务的数据存储与单体应用程序的数据库同步。但是，这可能会导致数据冗余以及两个数据存储之间的最终一致性，因此我们建议您将其视为战术解决方案，直到您可以建立长期解决方案，例如数据湖。

实施

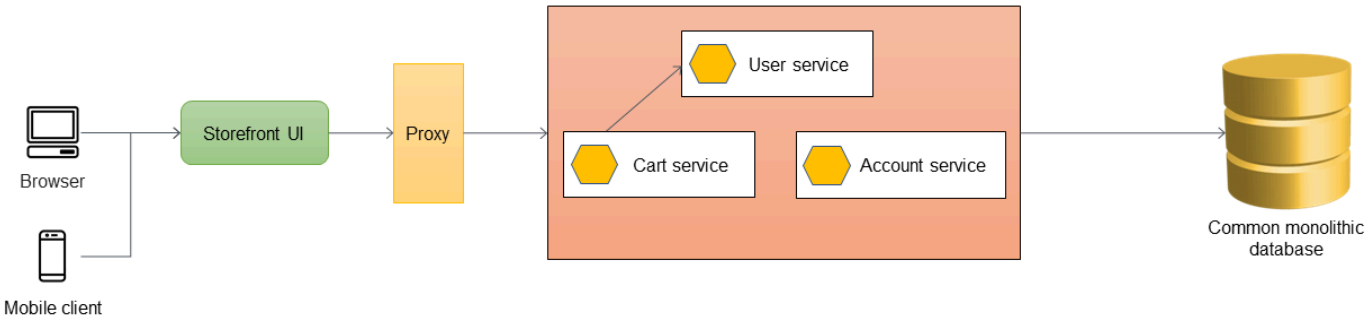
在 strangler fig 模式中，您可以将特定功能替换为新的服务或应用程序，一次一个组件。代理层会拦截发送到单体应用程序的请求，并将这些请求路由到遗留系统或新系统。由于代理层会将用户路由到正确的应用程序，因此您可以向新系统添加功能，同时确保单体继续运行。新系统最终取代了旧系统的所有功能，您可以将其停用。

架构简析

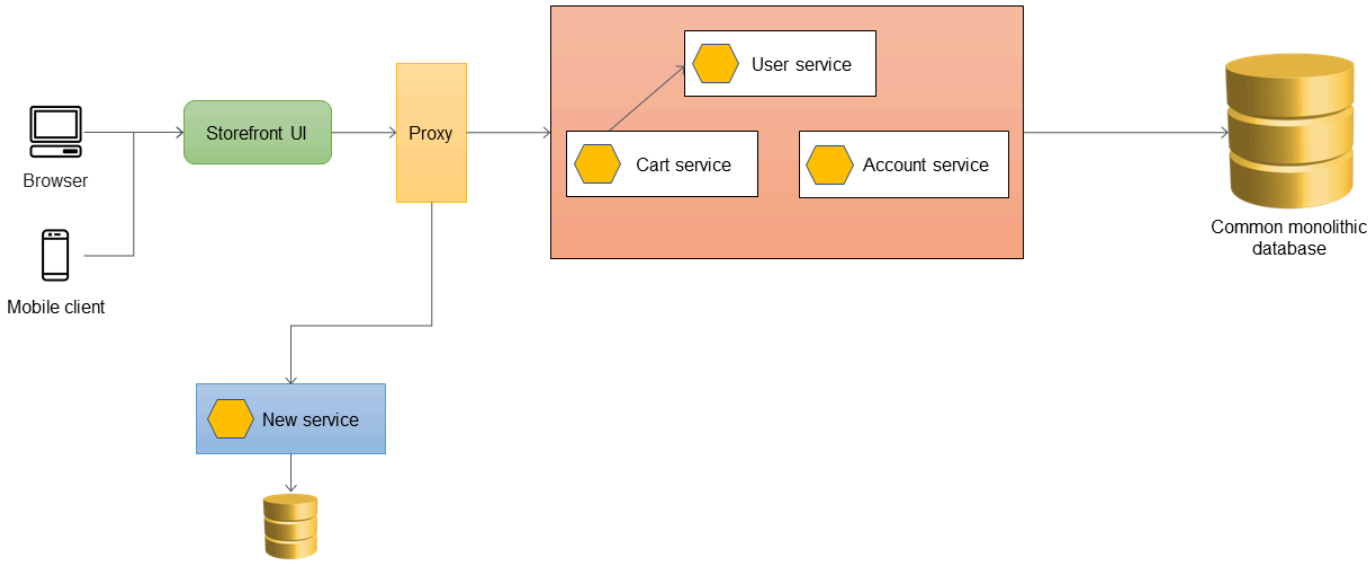
在下图中，单体应用程序具有三项服务：用户服务、购物车服务和账户服务。购物车服务依赖于用户服务，应用程序使用单体关系数据库。



第一步是在店铺界面和单体应用程序之间添加代理层。开始时，代理会将所有流量路由到单体应用程序。

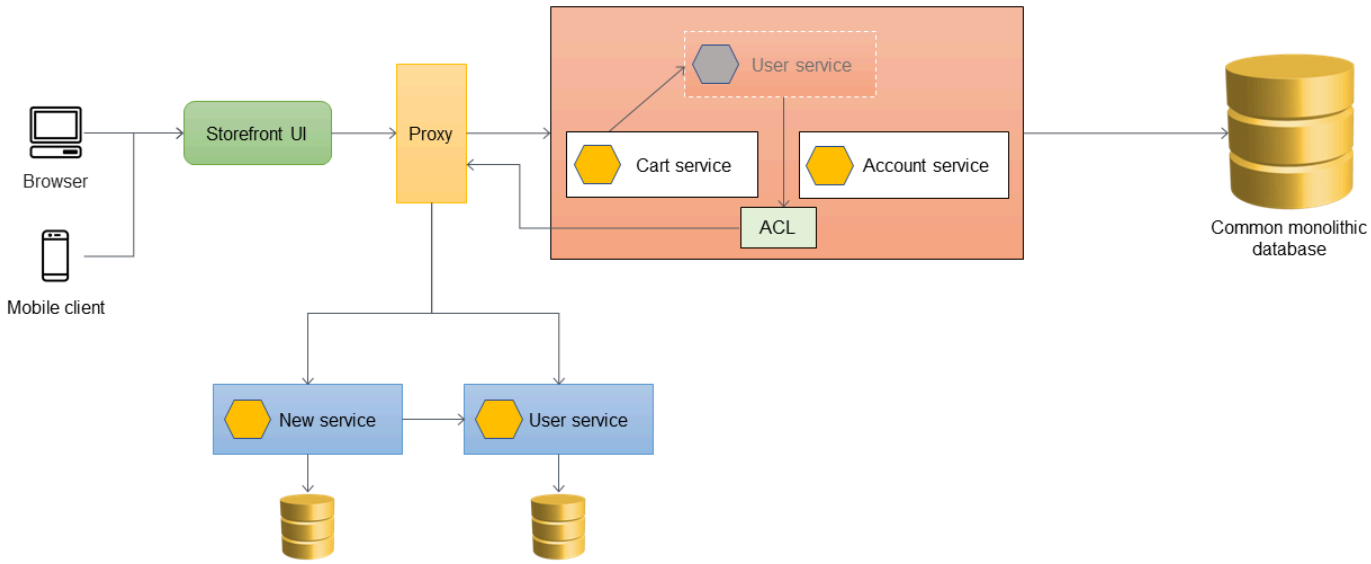


当您想向应用程序添加新功能时，可以将它们作为新的微服务来实现，而不是向现有的单体添加功能。但是，为了确保应用程序的稳定性，您需继续修复单体中的错误。在下图中，代理层根据 API URL 将调用路由到单体或新的微服务。



添加防腐层

在以下架构中，用户服务已迁移到微服务。购物车服务调用用户服务，但在单体中不再提供该实现。此外，新迁移的服务的接口可能与其在单体应用程序内部的先前接口不匹配。为了应对这些更改，您需要实施 ACL。在迁移过程中，当单体内的功能需要调用作为微服务迁移的功能时，ACL 会将调用转换为新接口并将其路由到相应的微服务。

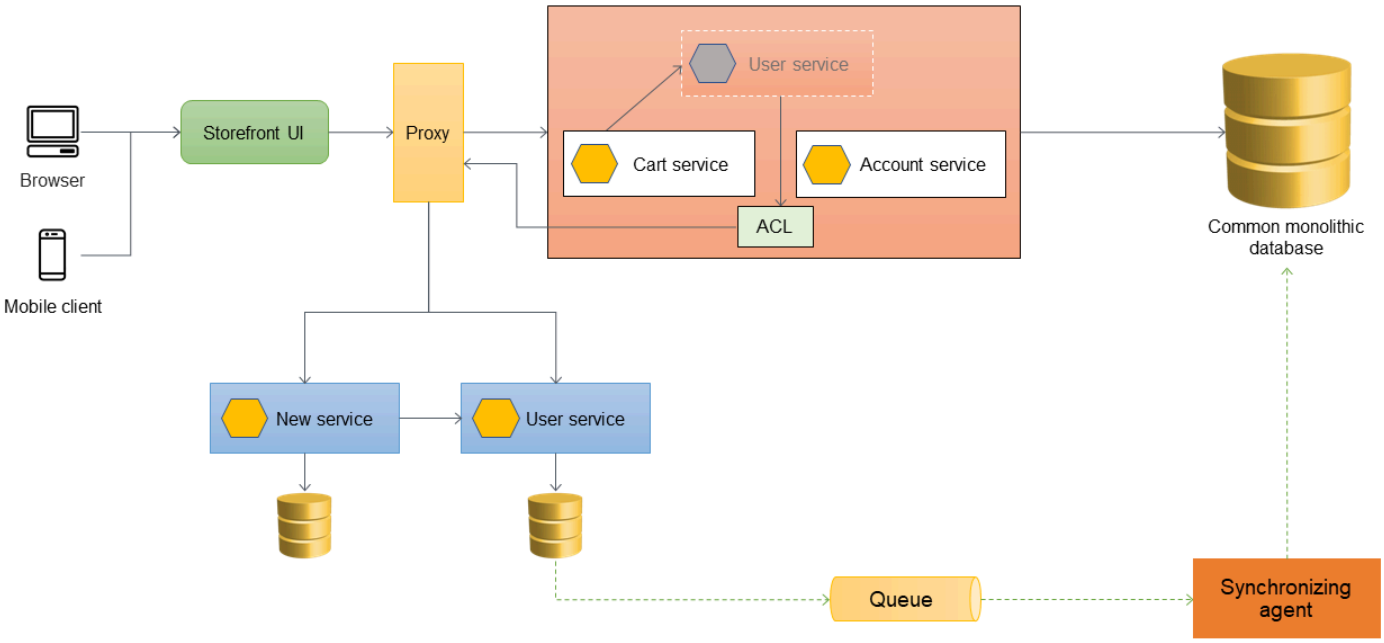


您可以在单体应用程序内部实现 ACL，将其作为特定于已迁移服务的类别；例如 `UserServiceFacade` 或 `UserServiceAdapter`。在所有从属服务都迁移到微服务架构后，必须停用 ACL。

使用 ACL 时，购物车服务仍会调用单体内的用户服务，而用户服务通过 ACL 将呼叫重定向到微服务。购物车服务仍应在不知道微服务迁移的情况下调用用户服务。此松耦合是减少回归和业务中断所必需的。

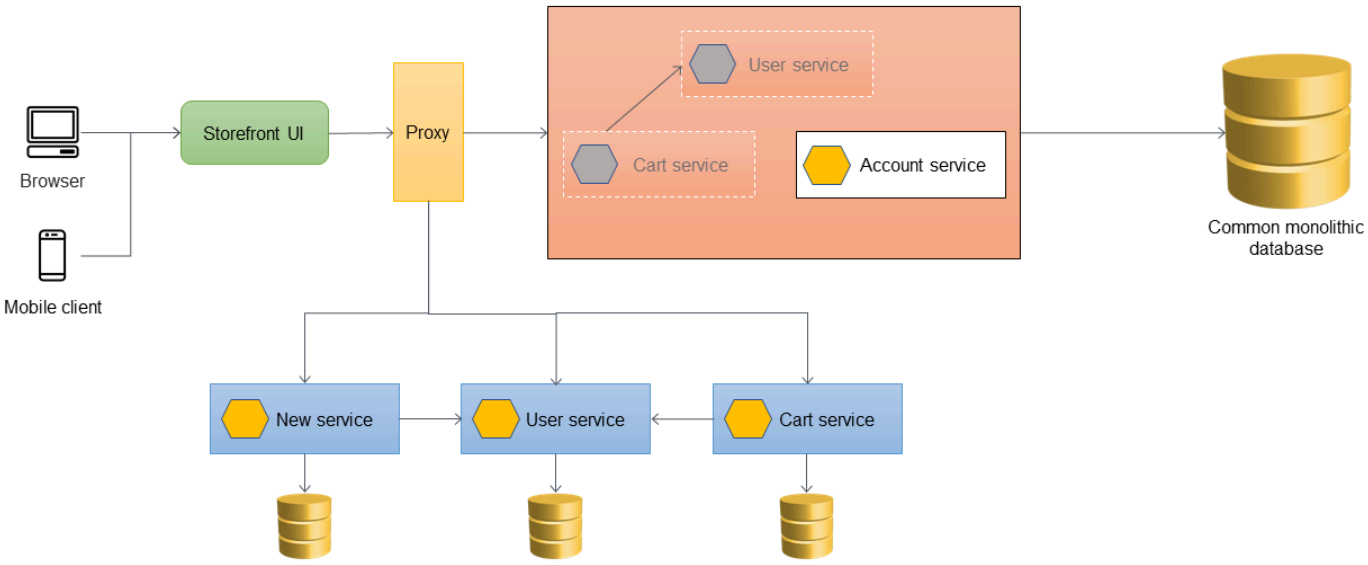
处理数据同步

作为最佳实践，微服务应拥有自己的数据。用户服务将其数据存储在自己的数据存储中。它可能需要将数据与单体数据库同步，以处理报告等依赖关系，并支持尚未准备好直接访问微服务的下游应用程序。单体应用程序可能还需要其他尚未迁移到微服务的功能和组件的数据。因此，需要在新的微服务和单体之间进行数据同步。要同步数据，您可以在用户微服务和单体数据库之间引入同步代理，如下图所示。每当用户微服务的数据库更新时，用户微服务都会向队列发送一个事件。同步代理会侦听队列，并持续更新单体数据库。对于正在同步的数据，单体数据库中的数据最终会保持一致。

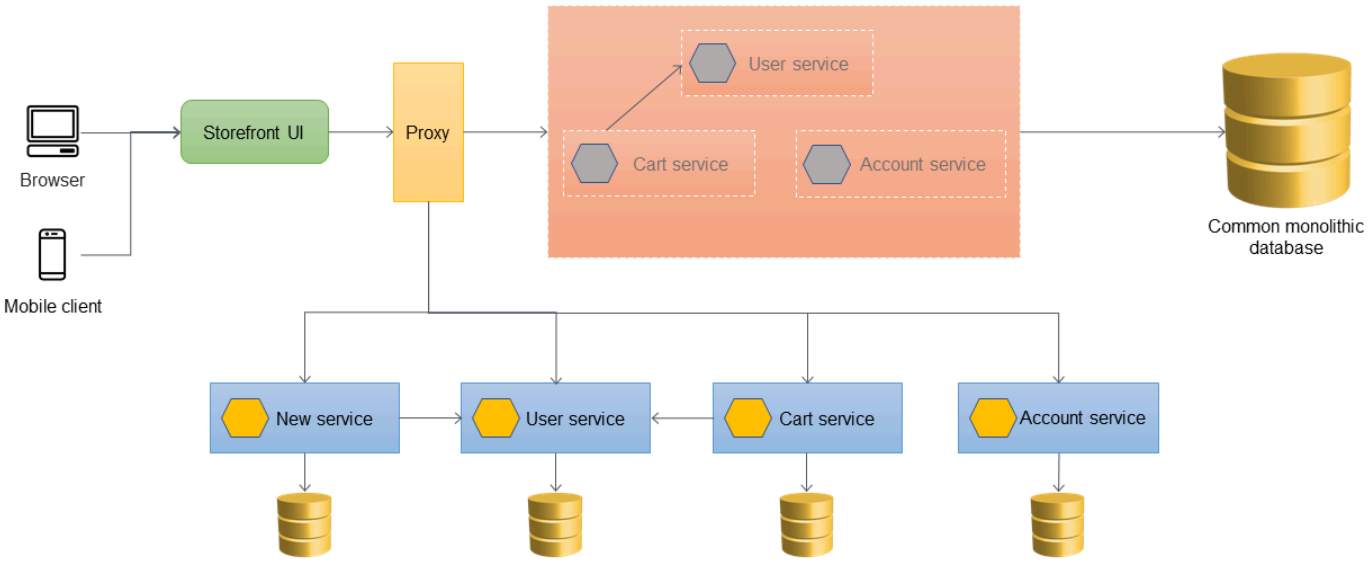


迁移其他服务

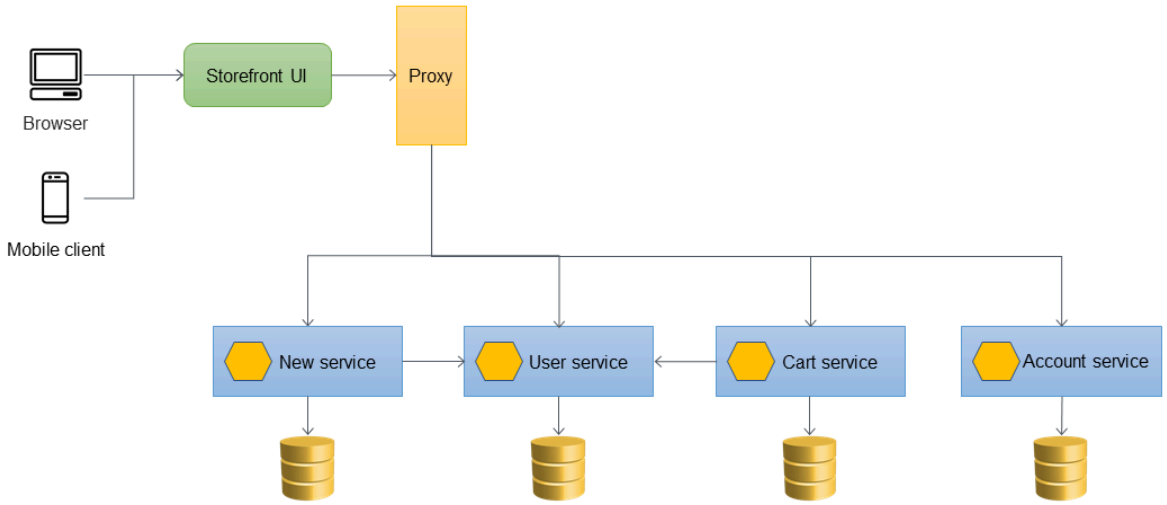
当购物车服务从单体应用程序中迁移出来时，其代码会修改为直接调用新服务，因此 ACL 不再路由这些调用。下图阐明了此架构。



下图显示了最终绞杀状态，其中所有服务都已从单体中迁出，仅剩单体的骨架。历史数据可以迁移到各个服务拥有的数据存储。ACL 可以删除，此阶段单体即可停用。



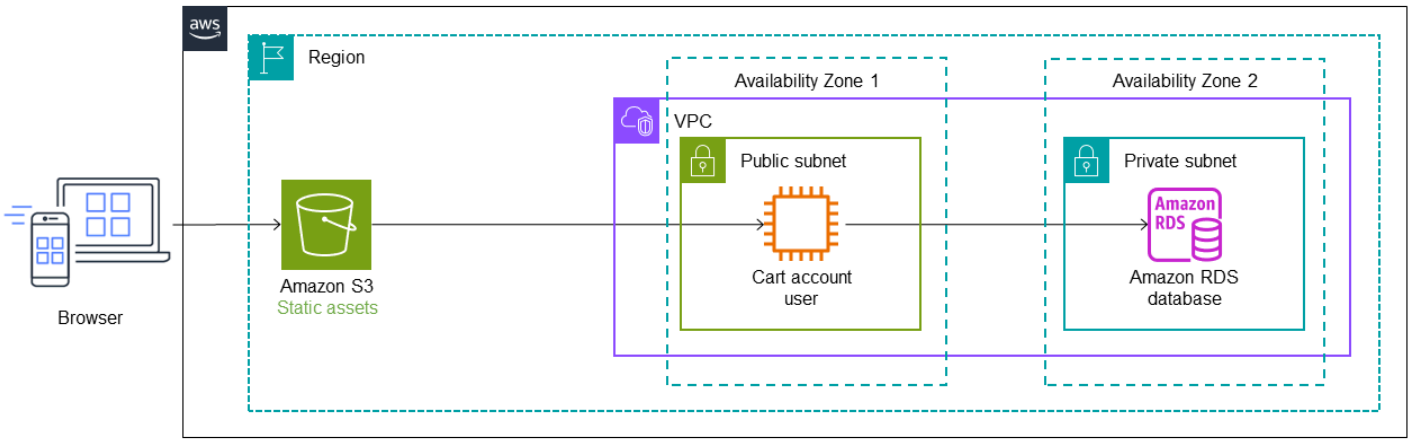
下图显示了单体应用程序停用后的最终架构。您可以根据应用程序的要求通过基于资源的 URL（例如 `http://www.storefront.com/user`）或通过其自己的域（例如 `http://user.storefront.com`）托管各个微服务。有关使用主机名和路径 APIs 向上游使用者公开 HTTP 的主要方法的更多信息，请参阅 API 路由模式部分。



使用 AWS 服务实施

使用 API Gateway 作为应用程序代理

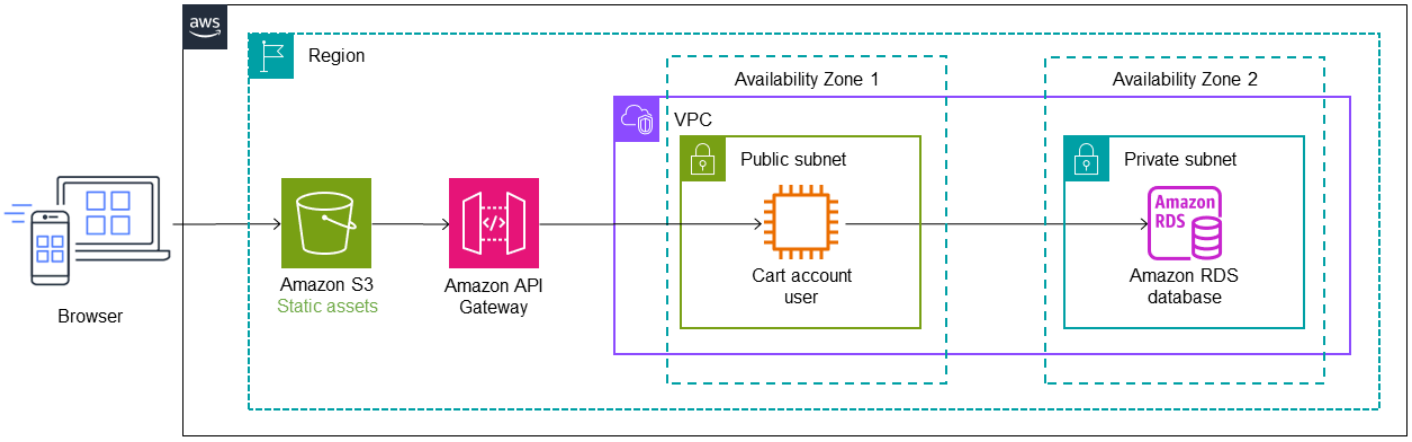
下图显示了单体应用程序的初始状态。假设它是 AWS 通过使用 lift-and-shift策略迁移到的，因此它在亚马逊弹性计算云 (Amazon EC2) 实例上运行，并使用亚马逊关系数据库服务 (Amazon RDS) 数据库。为简单起见，该架构使用具有一个私有子网和一个公有子网的单个虚拟私有云（VPC），假设微服务最初将部署在同一个 AWS 账户内。（生产环境中的最佳实践是使用多账户架构来确保部署的独立性。）EC2 实例驻留在公有子网中的单个可用区中，RDS 实例驻留在私有子网中的单个可用区中。亚马逊简单存储服务 (Amazon S3) S ervice 为 JavaScript网站存储静态资产，例如、CSS 和 React 文件。



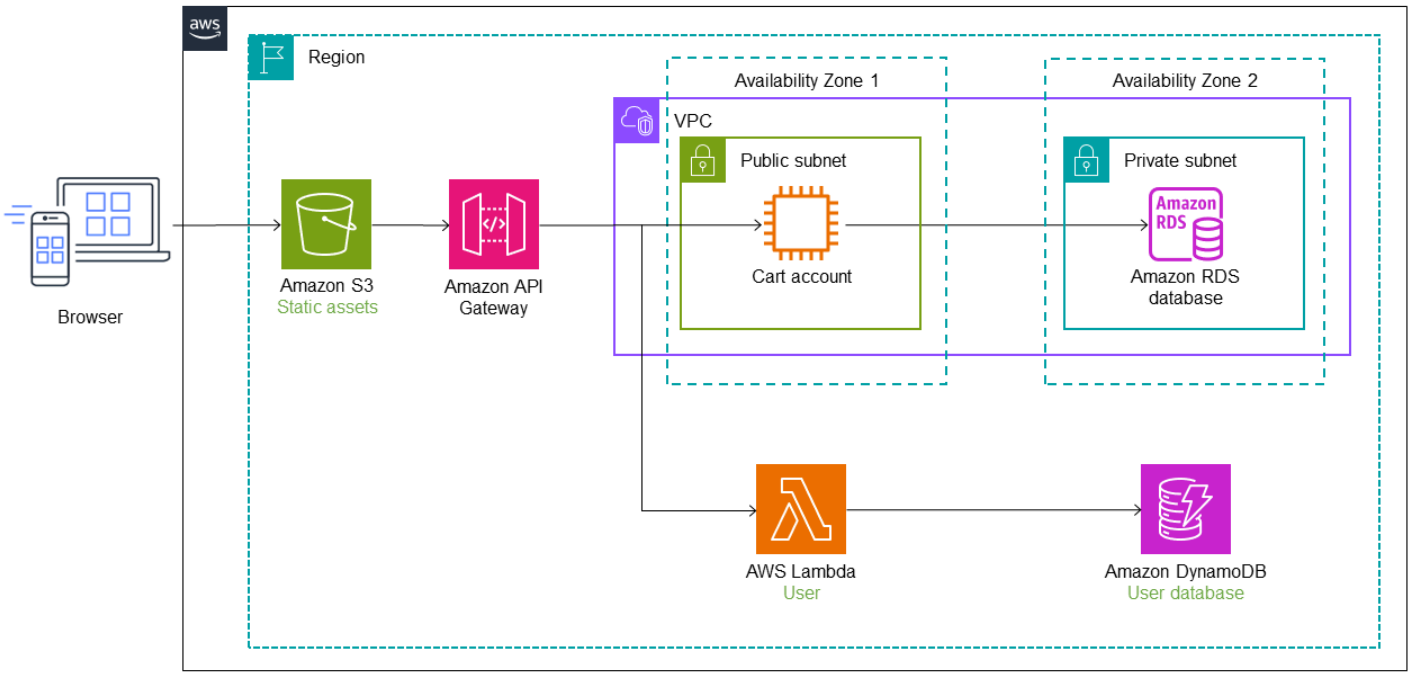
在以下架构中，AWS Migration Hub Refactor Spaces 在单体应用程序前部署 Amazon API Gateway。Refactor Spaces 会在您的账户内部创建重构基础设施，而 API Gateway 充当将调用路由到

单体的代理层。最初，所有调用都通过代理层路由到单体应用程序。如前所述，代理层可能成为单点故障。但是，使用 API Gateway 作为代理可降低风险，因为它是一项无服务器的多可用区服务。

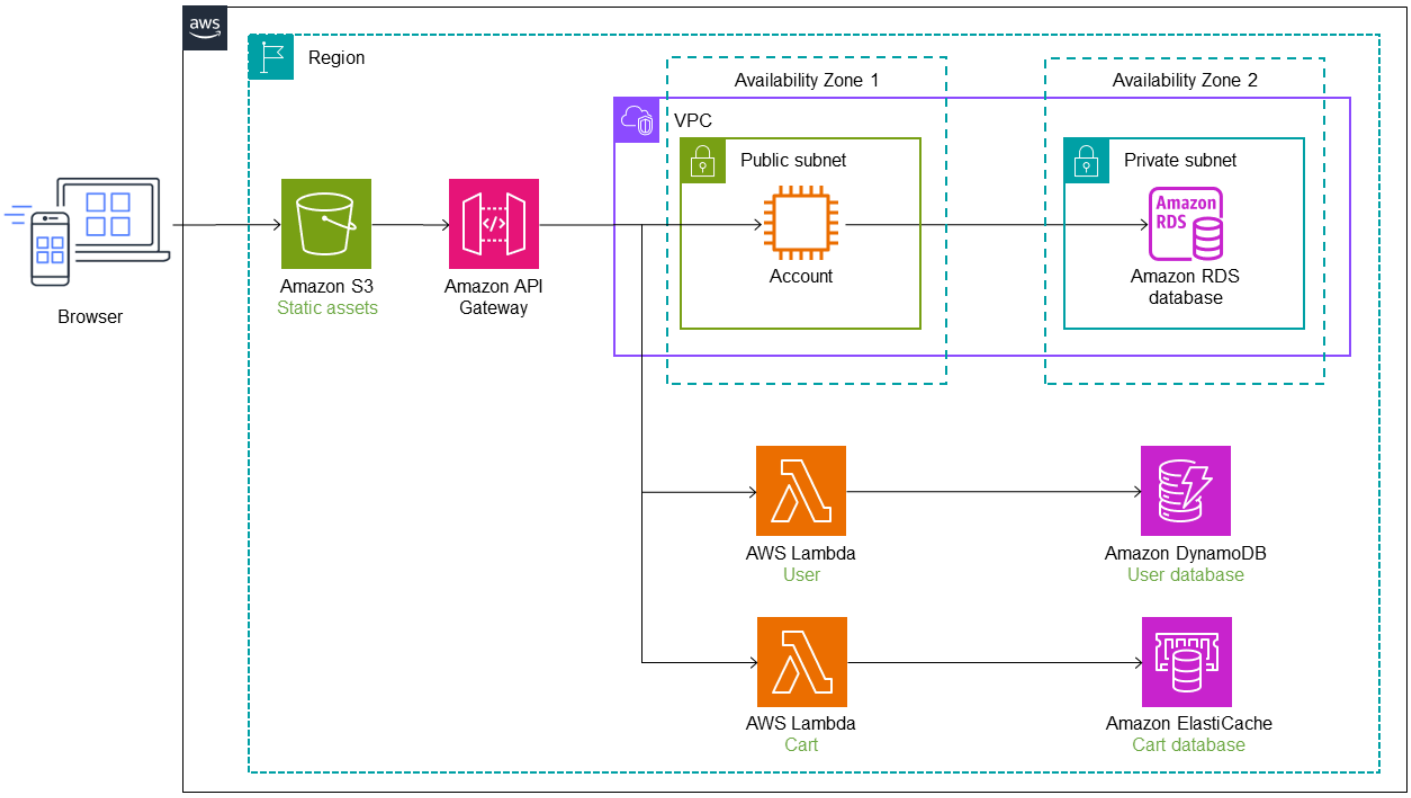
> **Note**
>
> AWS Migration Hub Refactor Spaces 自 2025 年 11 月 7 日起，不再向新客户开放。要获得类似的功能 AWS Migration Hub Refactor Spaces，请探索AWS Transform。



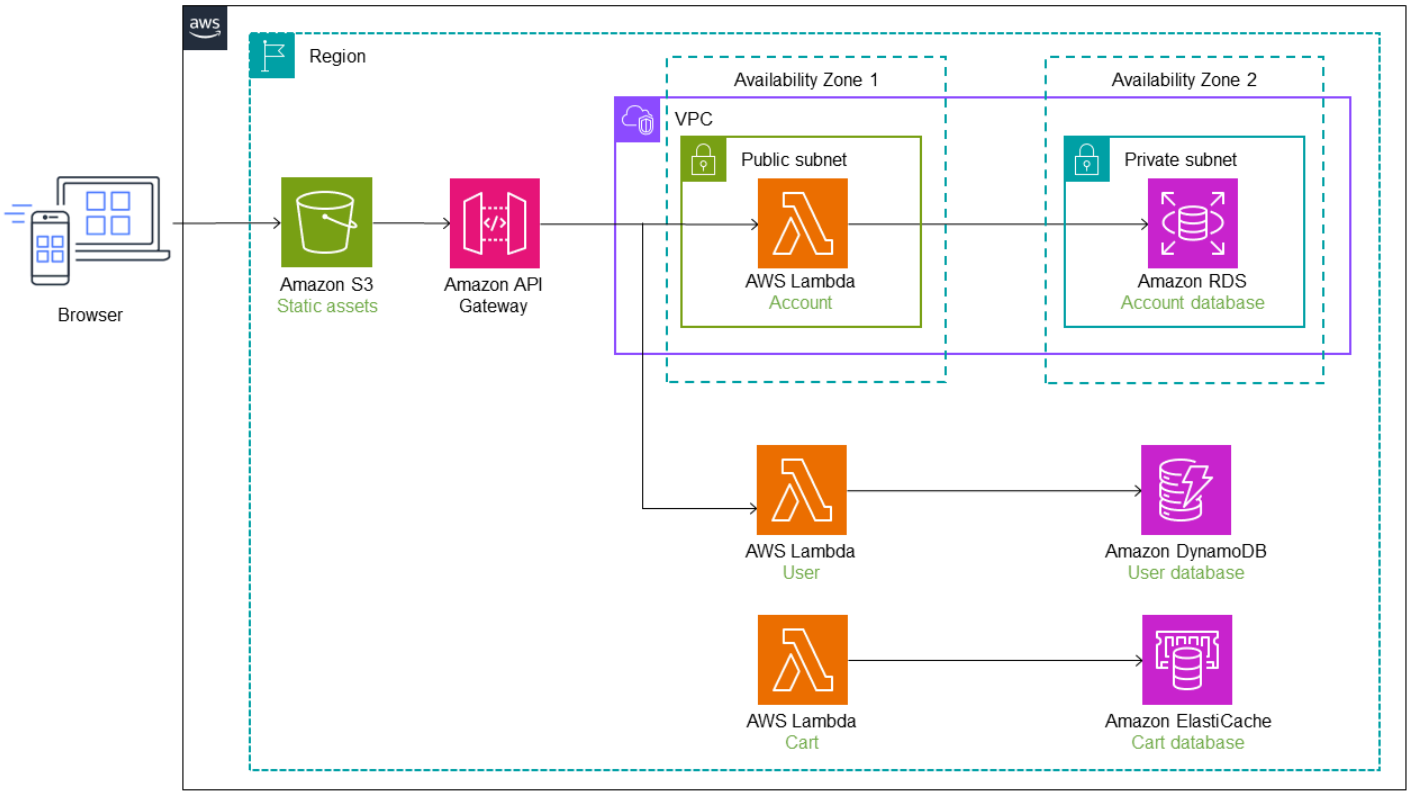
用户服务迁移到 Lambda 函数中，Amazon DynamoDB 数据库存储其数据。Lambda 服务端点和默认路由已添加到 Refactor Spaces，且 API Gateway 会自动配置为将调用路由到 Lambda 函数。



在下图中，购物车服务也已从单体迁移到 Lambda 函数。Refactor Spaces 中添加了额外的路由和服务端点，流量会自动割接到 `Cart` Lambda 函数。Lambda 函数的数据存储由亚马逊管理。ElastiCache单体应用程序仍与 Amazon RDS 数据库一起保留在 EC2 实例中。



在下图中，最后一个服务（账户）从单体迁移到 Lambda 函数中。它继续使用原始 Amazon RDS 数据库。新架构现在具有三个带有独立数据库的微服务。每种服务均使用不同类型的数据库。使用专用数据库来满足微服务的特定需求这一概念称为多语言持久性。Lambda 函数也可以用不同的编程语言实现，具体取决于使用案例。在重构期间，Refactor Spaces 会自动将流量割接和路由到 Lambda。这可以为您的构建者节省架构、部署和配置路由基础设施所需的时间。

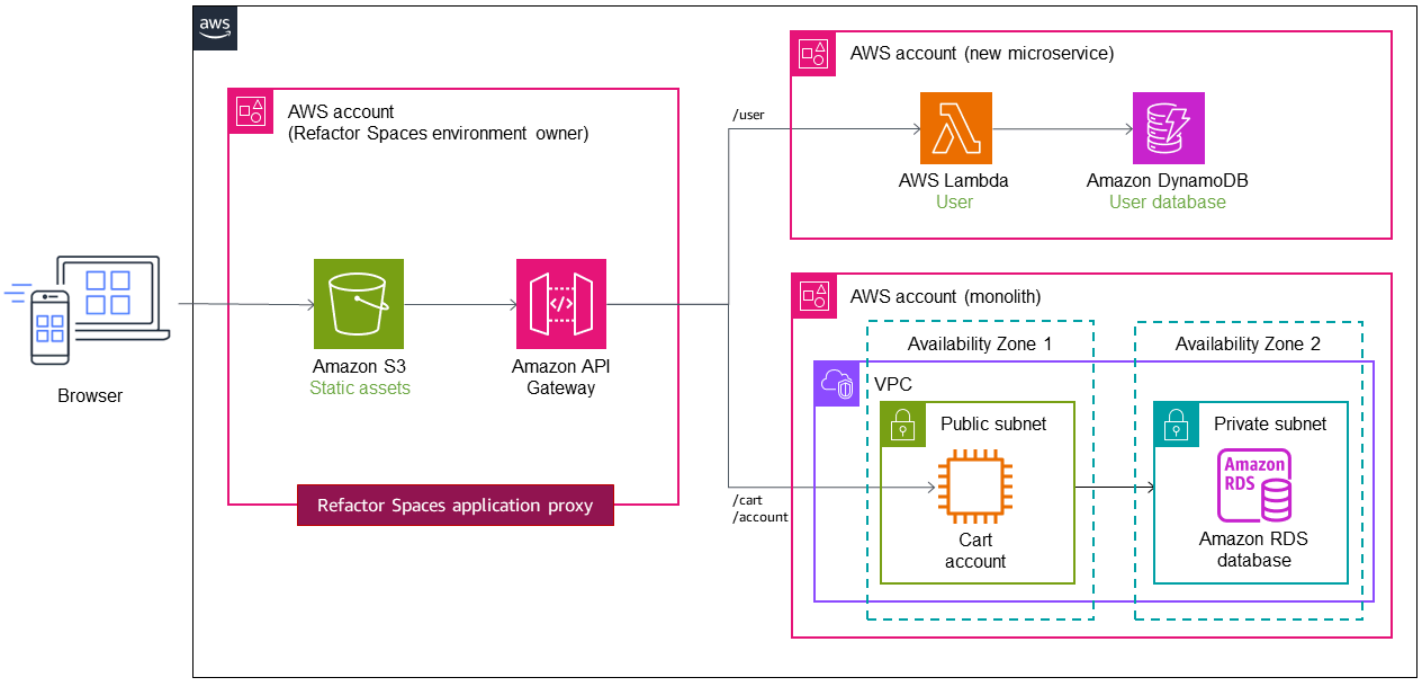


使用多个账户

在之前的实现中，我们使用了具有私有子网和公有子网的单个 VPC 作为单体应用程序，为了简单起见，我们在同一个 AWS 账户 内部署了微服务。但是，在现实场景中，这种情况很少见，在这些场景中， AWS 账户 为了独立于部署，微服务通常被分成多个部署。在多账户结构中，您需要配置从单体到不同账户的新服务的路由流量。

重构空间可帮助您创建和配置 AWS 基础架构，以便将 API 调用从单体应用程序中路由出去。作为应用程序资源的一部分，Refactor Spaces 在您的 AWS 账户内编排 API Gateway、网络负载均衡器和基于资源的 AWS Identity and Access Management （ IAM ）策略。您可以透明地将单个 AWS 账户 或多个账户中的新服务添加到外部 HTTP 端点。所有这些资源都在您的内部编排，可以在 AWS 账户 部署后进行自定义和配置。

假设用户和购物车服务部署到两个不同的账户，如下图所示。使用 Refactor Spaces 时，您只需要配置服务端点和路由。Refactor Spaces 可自动执行 API Gateway—Lambda 集成和 Lambda 资源策略的创建，因此您可以专注于安全地从单体上重构服务。



有关使用 Refactor Spaces 的视频教程，请参阅 Refactor Apps Incrementally with AWS Migration Hub Refactor Spaces。

研讨会

- Iterative App Modernization 讲习会

参考博客文章

- AWS Migration Hub Refactor Spaces
- Deep Dive on an AWS Migration Hub Refactor Spaces
- 部署管线参考架构和参考实现

相关内容

- API 路由模式
- Refactor Spaces 文档

事务发件箱模式

意图

事务发件箱模式解决了分布式系统中的双重写入操作问题，此问题出现于单个操作同时涉及数据库写入操作和消息或事件通知时。当应用程序向两个不同的系统写入数据时，便会发生双重写入操作；例如，当微服务需要在数据库中持久化数据并发送消息以通知其他系统时。其中一个操作失败便可能会导致数据不一致。

动机

当微服务在数据库更新后发送事件通知时，这两个操作应以原子方式运行，从而确保数据一致性和可靠性。

- 如果数据库更新成功但事件通知失败，则下游服务将不知道有发生更改，系统可能会进入不一致的状态。
- 如果数据库更新失败但发送了事件通知，则数据可能会损坏，由此可能会影响系统的可靠性。

适用性

在以下情况使用事务发件箱模式：

- 您正在构建事件驱动的应用程序，其中的数据库更新会启动事件通知。
- 您需要确保涉及两项服务的操作的原子性。
- 您想实现事件溯源模式。

问题和注意事项

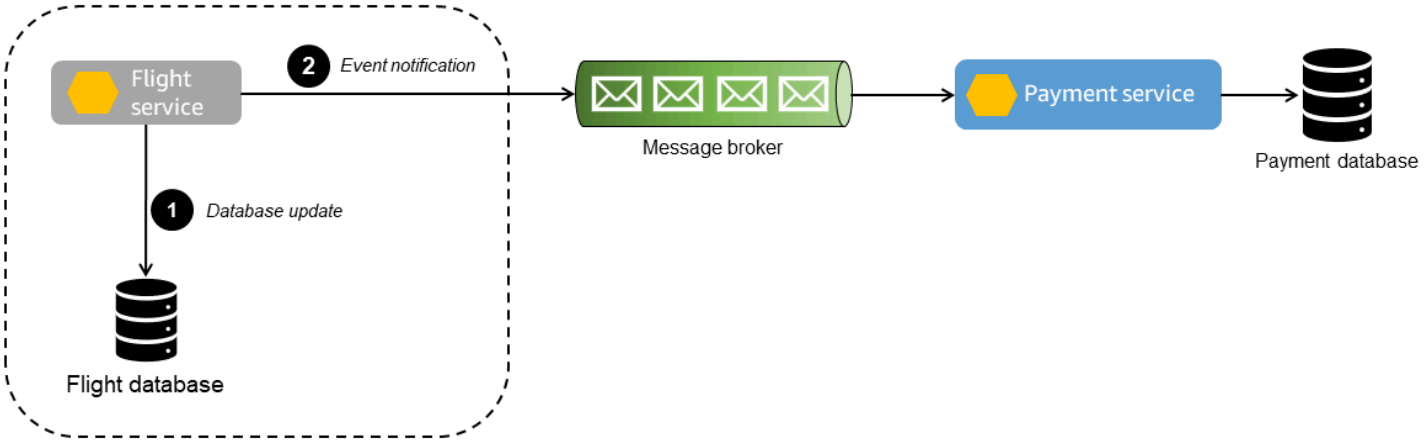
- 重复消息：事件处理服务可能会发送重复的消息或事件，因此建议您通过跟踪已处理的消息来令服务的使用具有幂等性。
- 通知顺序：按照服务更新数据库的同一顺序发送消息或事件。这对于事件源模式至关重要，在这种模式中，您可以使用事件存储来 point-in-time恢复数据存储。如果顺序不正确，则可能会影响数据的质量。如果未持久化通知顺序，则最终一致性和数据库回滚可能会将问题复杂化。
- 事务回滚：如果事务已回滚，请勿发送事件通知。

- 服务级事务处理：如果事务跨越需要数据存储更新的服务，则请使用 saga 编排模式来保持数据存储中的数据完整性。

实施

高级架构

以下序列图显示了双重写入操作期间发生的事件顺序。



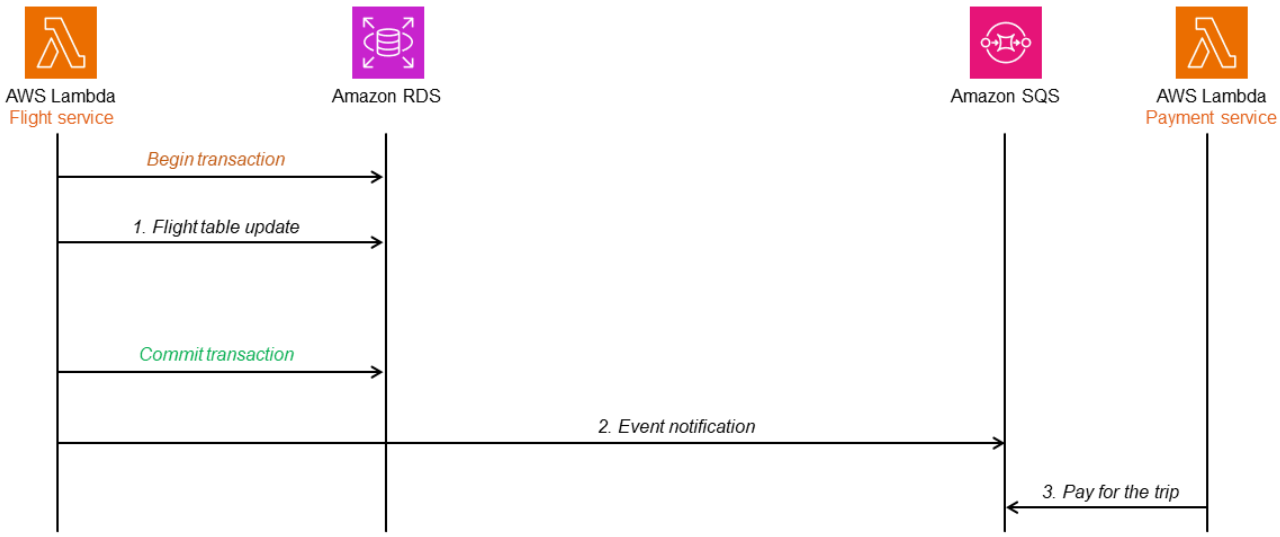
1. 航班服务写入数据库，并向付款服务发送出事件通知。
2. 消息代理将消息和事件传送至付款服务。消息代理中的任何故障都会导致付款服务无法接收更新。

如果航班数据库更新失败但通知已发出，则付款服务将根据事件通知处理付款。此操作将导致下游数据不一致。

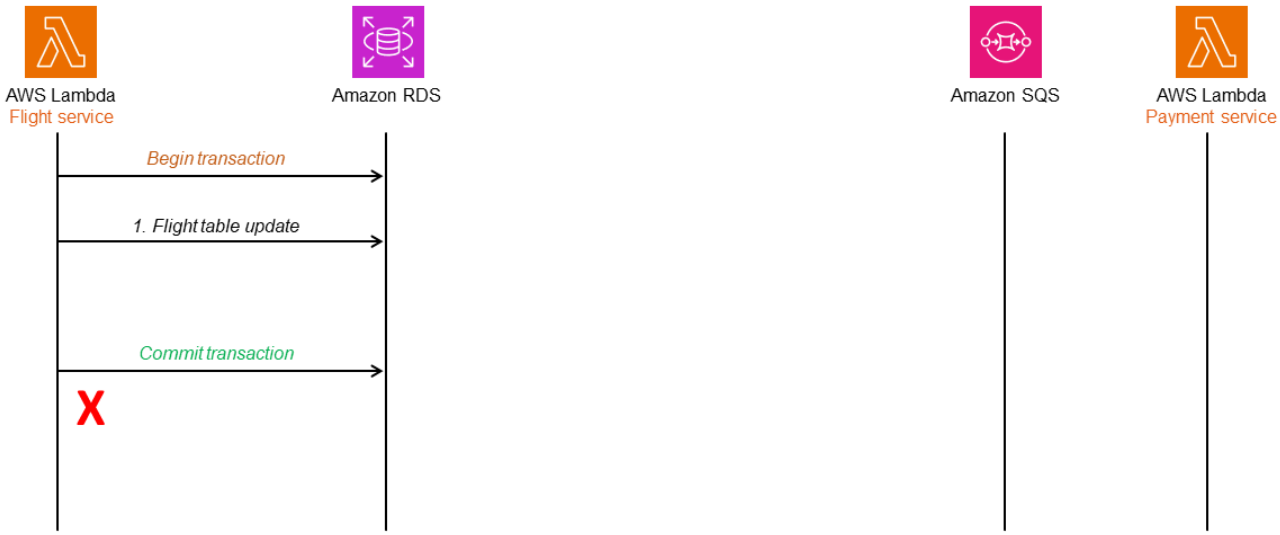
使用 AWS 服务实施

为了演示序列图中的模式，我们将使用以下 AWS 服务，如下图所示。

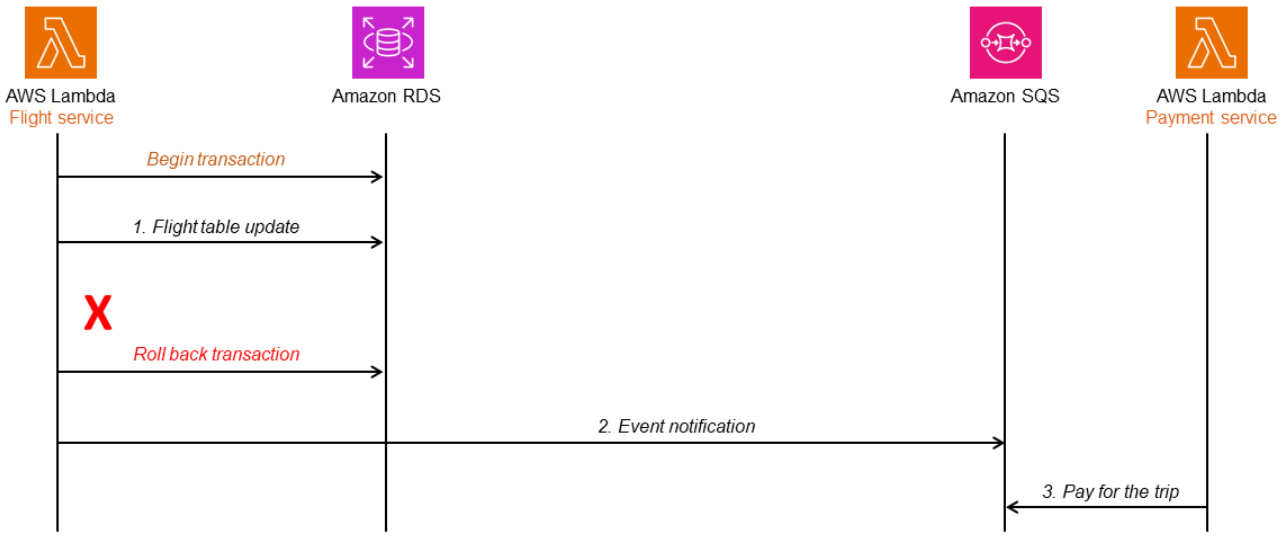
- 微服务是通过使用 AWS Lambda 实现的。
- 主数据库由 Amazon Relational Database Service（Amazon RDS）管理。
- Amazon Simple Queue Service（Amazon SQS）充当接收事件通知的消息代理。



如果航班服务在提交事务后出现故障，则可能导致无法发送事件通知。



但是，事务可能会失败并回滚，但事件通知可能仍会发送，从而导致付款服务处理付款。



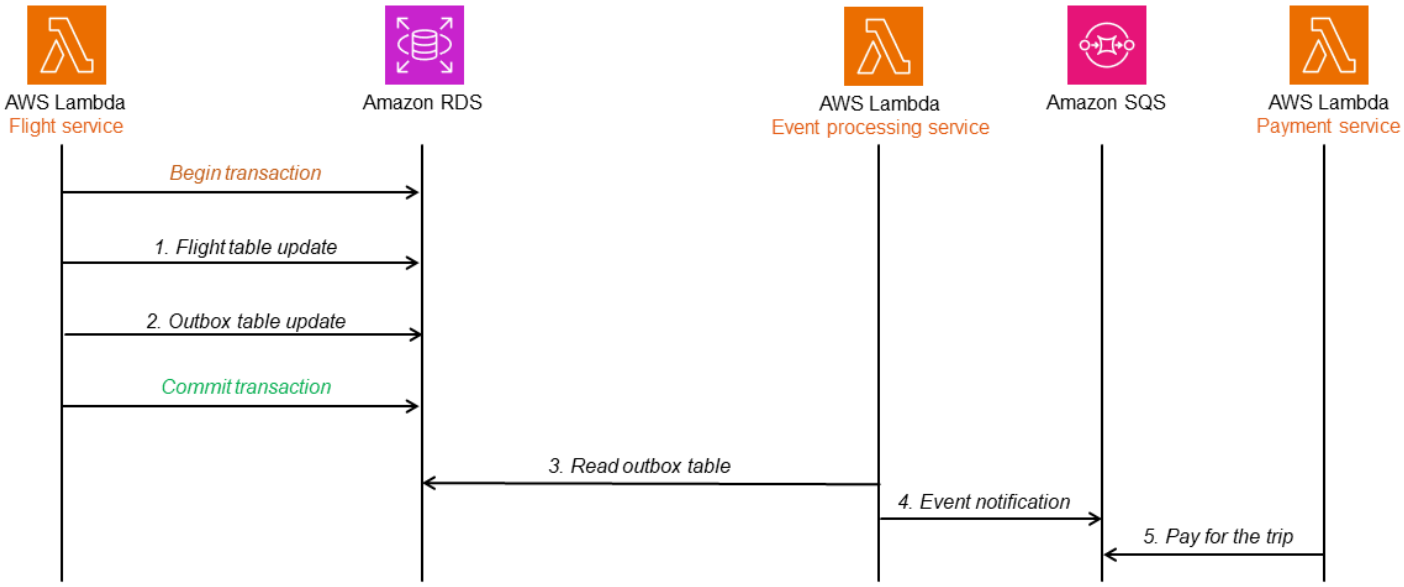
要解决此问题，您可以使用发件箱表或更改数据捕获（CDC）。以下各部分将讨论这两个选项，以及如何使用亚马逊云科技服务实现它们。

将发件箱表与关系数据库配合使用

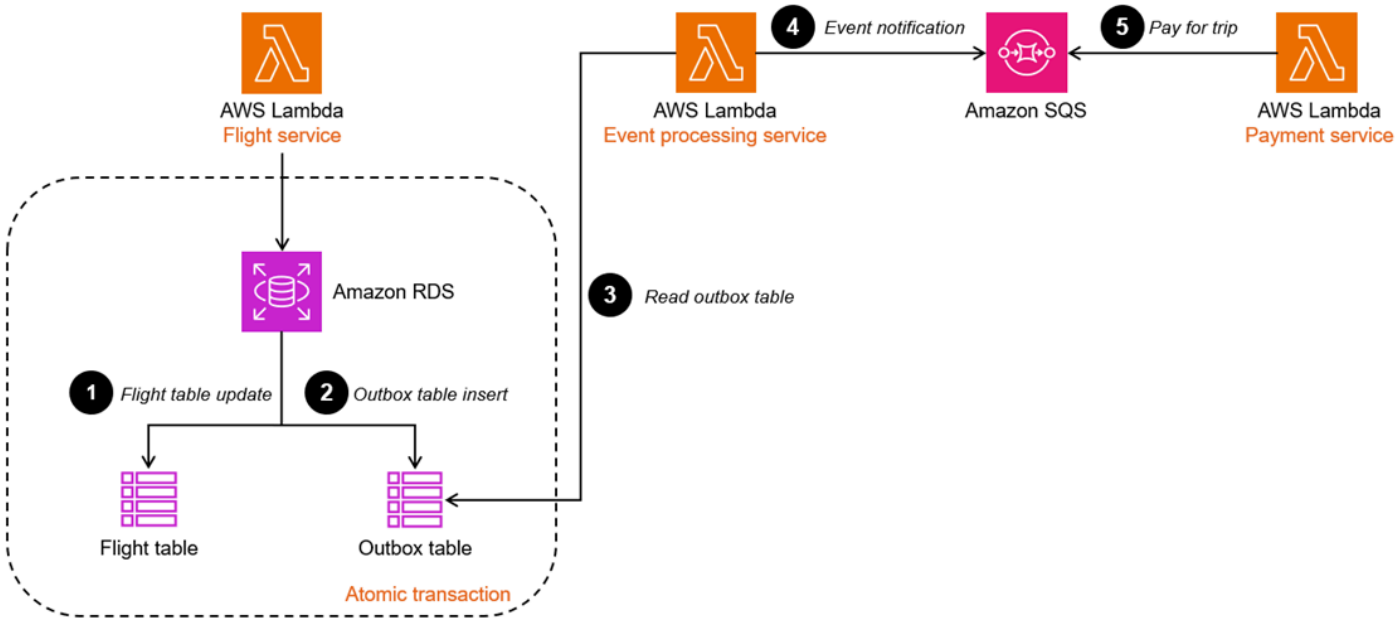
发件箱表存储来自航班服务的所有事件，带有时间戳和序列号。

当航班表更新时，发件箱表也会在同一事务中更新。另一项服务（例如事件处理服务）从发件箱表中读取信息并将事件发送到 Amazon SQS。Amazon SQS 会向付款服务发送有关该事件的消息以供进一步处理。Amazon SQS 标准队列可保证消息至少传送一次，且不会丢失。但是，当您使用 Amazon SQS 标准队列时，同一条消息或事件可能会多次传送，因此您应确保事件通知服务是幂等性的（也就是说，多次处理同一条消息不会产生不利影响）。如果您要求消息只处理一次，并采用消息排序，则可以使用 Amazon SQS 先入先出 (FIFO) 队列。

如果航班表更新失败，或发件箱表更新失败，则会回滚整个事务，因此不会出现下游数据不一致的情况。



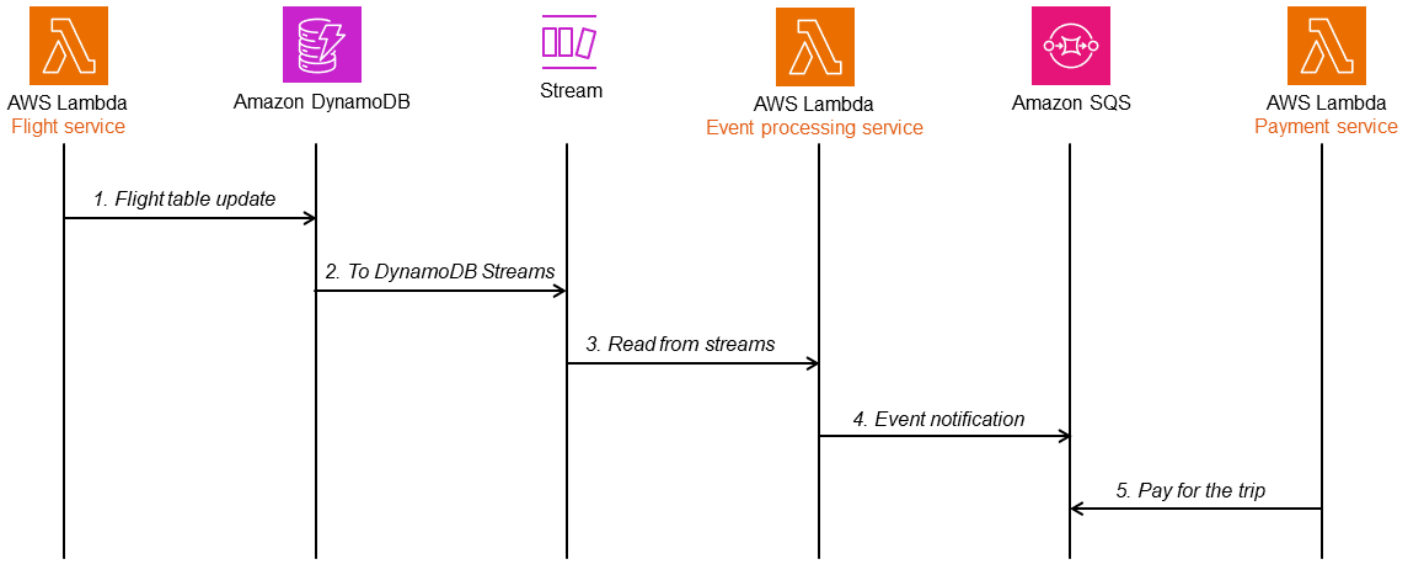
在下图中，事务发件箱架构是使用 Amazon RDS 数据库实现的。当事件处理服务读取发件箱表时，它只识别已提交（成功）事务中的那些行，然后将事件的消息放入 SQS 队列中，由付款服务读取该队列以供进一步处理。这种设计解决了双重写入操作问题，并通过使用时间戳和序列号来持久化消息和事件的顺序。



使用更改数据捕获（CDC）

某些数据库支持发布项目级修改，以捕获已更改的数据。您可以识别已更改的项目，并相应地发送事件通知。这样可以节省创建用于跟踪更新的另一个表的开销。航班服务发起的事件存储在同一项目的另一个属性中。

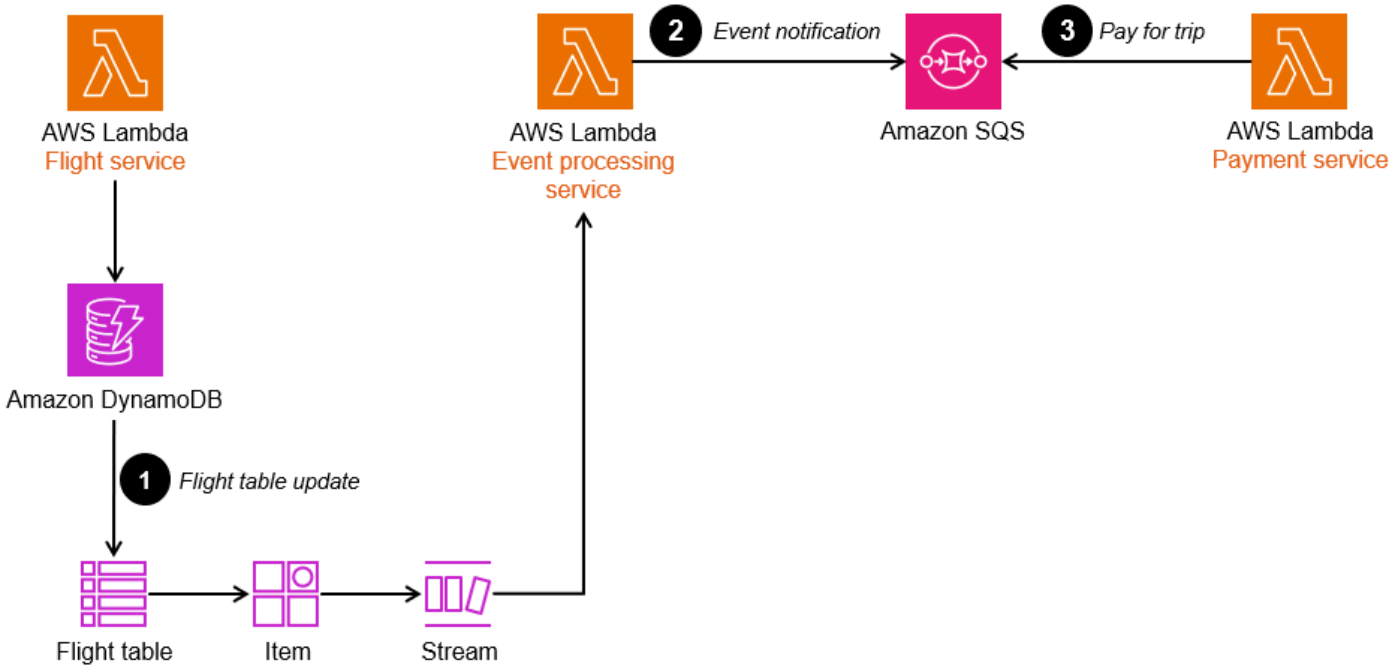
Amazon DynamoDB 是一个键/值 NoSQL 数据库，支持 CDC 更新。在下面的序列图中，DynamoDB 发布了对 Amazon DynamoDB Streams 的项目级修改。事件处理服务从流中读取数据，并将事件通知发布到付款服务以供进一步处理。



DynamoDB Streams 使用时间排序序列捕获与 DynamoDB 表中项目级更改相关的信息流。

您可以通过在 DynamoDB 表上启用流来实现事务发件箱模式。事件处理服务的 Lambda 函数与这些流存在关联。

- 更新航班表后，DynamoDB Streams 会捕获已更改的数据，事件处理服务会轮询流以查找新记录。
- 当新的流记录可用时，Lambda 函数会同步将事件的消息放入 SQS 队列中，便于进一步处理。您可以根据需要向 DynamoDB 项目添加属性，以捕获时间戳和序列号，从而提高实施的稳健性。



代码示例

使用发件箱表

本节中的代码示例显示了如何使用发件箱表实现事务发件箱模式。要查看完整的代码，请参阅此示例的GitHub存储库。

以下代码片段在单个事务中将数据库中的 `Flight` 实体和 `Flight` 事件保存在其各自的表中。

```
@PostMapping("/flights")
    @Transactional
    public Flight createFlight(@Valid @RequestBody Flight flight) {
        Flight savedFlight = flightRepository.save(flight);
        JsonNode flightPayload = objectMapper.convertValue(flight, JsonNode.class);
        FlightOutbox outboxEvent = new FlightOutbox(flight.getId().toString(),
 FlightOutbox.EventType.FLIGHT_BOOKED,
                flightPayload);
        outboxRepository.save(outboxEvent);
        return savedFlight;
    }
```

另一项服务负责定期扫描发件箱表中是否有新事件，将其发送到 Amazon SQS，如果 Amazon SQS 成功响应，则将其从表格中删除。轮询速率可在 `application.properties` 文件中配置。

```
@Scheduled(fixedDelayString = "${sqs.polling_ms}")
    public void forwardEventsToSQS() {
        List<FlightOutbox> entities =
 outboxRepository.findAllByOrderByIdAsc(Pageable.ofSize(batchSize)).toList();
        if (!entities.isEmpty()) {
            GetQueueUrlRequest getQueueRequest = GetQueueUrlRequest.builder()
                    .queueName(sqsQueueName)
                    .build();
            String queueUrl = this.sqsClient.getQueueUrl(getQueueRequest).queueUrl();
            List<SendMessageBatchRequestEntry> messageEntries = new ArrayList<>();
            entities.forEach(entity ->
 messageEntries.add(SendMessageBatchRequestEntry.builder()
                    .id(entity.getId().toString())
                    .messageGroupId(entity.getAggregateId())
                    .messageDeduplicationId(entity.getId().toString())
                    .messageBody(entity.getPayload().toString())
                    .build())
            );
            SendMessageBatchRequest sendMessageBatchRequest =
 SendMessageBatchRequest.builder()
                    .queueUrl(queueUrl)
                    .entries(messageEntries)
                    .build();
            sqsClient.sendMessageBatch(sendMessageBatchRequest);
            outboxRepository.deleteAllInBatch(entities);
        }
    }
```

使用更改数据捕获（CDC）

本节中的示例代码显示了如何使用 DynamoDB 的更改数据捕获（CDC）功能实现事务发件箱模式。要查看完整的代码，请参阅此示例的GitHub存储库。

以下 AWS Cloud Development Kit (AWS CDK) 代码片段创建一个 DynamoDB 航班表和 Amazon Kinesis 数据流（`cdcStream`），并将航班表配置为将其所有更新发送到该流。

```
Const cdcStream = new kinesis.Stream(this, 'flightsCDCStream', {
    streamName: 'flightsCDCStream'
})

const flightTable = new dynamodb.Table(this, 'flight', {
    tableName: 'flight',
```

```
    kinesisStream: cdcStream,
    partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
    }

});
```

以下代码片段和配置定义一个 spring cloud stream 函数，该函数在 Kinesis 流中获取更新并将这些事件转发到 SQS 队列以进行进一步处理。

```
applications.properties
spring.cloud.stream.bindings.sendToSQS-in-0.destination=${kinesisstreamname}
spring.cloud.stream.bindings.sendToSQS-in-0.content-type=application/ddb

QueueService.java
@Bean
public Consumer<Flight> sendToSQS() {
    return this::forwardEventsToSQS;
}

public void forwardEventsToSQS(Flight flight) {
    GetQueueUrlRequest getQueueRequest = GetQueueUrlRequest.builder()
            .queueName(sqsQueueName)
            .build();
    String queueUrl = this.sqsClient.getQueueUrl(getQueueRequest).queueUrl();
    try {
        SendMessageRequest send_msg_request = SendMessageRequest.builder()
                .queueUrl(queueUrl)
                .messageBody(objectMapper.writeValueAsString(flight))
                .messageGroupId("1")
                .messageDeduplicationId(flight.getId().toString())
                .build();
        sqsClient.sendMessage(send_msg_request);
    } catch (IOException | AmazonServiceException e) {
        logger.error("Error sending message to SQS", e);
    }
}
```

GitHub 存储库

有关此模式示例架构的完整实现，请参见 GitHub存储库，网址为https://github.com/aws-samples/transactional-outbox-pattern。

资源

参考

- AWS 架构中心
- AWS 开发人员中心
- Amazon Builders Library

工具

- AWS Well-Architected Tool
- AWS App2Container
- AWS Microservice Extractor for .NET

方法

- The Twelve-Factor App（Adam Wiggins 著，ePub 版）
- Nygard，Michael T. Release It!: Design and Deploy Production-Ready Software（第 2 版）北卡罗来纳州罗利：Pragmatic 出版社，2018 年。
- Polyglot Persistence（Martin Fowler 的博客文章）
- StranglerFigApplication（Martin Fowler 的博客文章）

文档历史记录

下表介绍了本指南的一些重要更改。如果您希望收到有关未来更新的通知，可以订阅 RSS 源。

变更	说明	日期
新模式	添加了两个新模式：六边形架构和分散-收集。	2024 年 5 月 7 日
新代码示例	向事务发件箱模式添加了更改数据捕获（CDC）使用案例的示例代码。	2024 年 2 月 23 日
新代码示例	• 使用示例代码更新了事务发件箱模式。 • 删除了关于编排和编配模式的部分，这些部分被 saga 编配和 saga 编排所取代。	2023 年 11 月 16 日
新模式	增加了三种新模式：saga 编配、发布订阅和事件溯源。	2023 年 11 月 14 日
更新	更新了 strangler fig 模式实施部分。	2023 年 10 月 2 日
初次发布	第一个版本包括八种设计模式：防腐层（ACL）、API 路由、断路器、编排和编配、退避重试、saga 编排、strangler fig 和事务发件箱。	2023 年 7 月 28 日

AWS 规范性指导词汇表

以下是 AWS 规范性指导提供的策略、指南和模式中的常用术语。若要推荐词条，请使用术语表末尾的提供反馈链接。

数字

7 R

将应用程序迁移到云中的 7 种常见迁移策略。这些策略以 Gartner 于 2011 年确定的 5 R 为基础，包括以下内容：

- 重构/重新架构：充分利用云原生功能来提高敏捷性、性能和可扩展性，以迁移应用程序并修改其架构。这通常涉及到移植操作系统和数据库。示例：将本地 Oracle 数据库迁移到 Amazon Aurora PostgreSQL 兼容版。
- 更换平台：将应用程序迁移到云中，并进行一定程度的优化，以利用云功能。示例：将本地 Oracle 数据库迁移到 AWS Cloud中的 Amazon Relational Database Service（Amazon RDS）for Oracle。
- 重新购买：转换到其他产品，通常是从传统许可转向 SaaS 模式。示例：将客户关系管理（CRM）系统迁移到 Salesforce.com。
- 重新托管（直接迁移）：将应用程序迁移到云中，无需进行任何更改即可利用云功能。示例：将本地 Oracle 数据库迁移到 AWS Cloud中 EC2 实例上的 Oracle。
- 重新放置（虚拟机监控器级直接迁移）：将基础设施迁移到云中，无需购买新硬件、重写应用程序或修改现有操作。您将服务器从本地平台迁移到同一平台的云服务中。示例：将Microsoft Hyper-V应用程序迁移到 AWS。
- 保留（重访）：将应用程序保留在源环境中。其中可能包括需要进行重大重构的应用程序，并且您希望将工作推迟到以后，以及您希望保留的遗留应用程序，因为迁移它们没有商业上的理由。
- 停用：停用或删除源环境中不再需要的应用程序。

A

ABAC

请参阅基于属性的访问控制。

抽象服务

请参阅托管服务。

ACID

请参阅原子性、一致性、隔离性、持久性。

主动-主动迁移

一种数据库迁移方法，在这种方法中，源数据库和目标数据库保持同步（通过使用双向复制工具或双写操作），两个数据库都在迁移期间处理来自连接应用程序的事务。这种方法支持小批量、可控的迁移，而不需要一次性割接。它比主动-被动迁移更灵活，但工作量更大。

主动-被动迁移

一种数据库迁移方法，在这种方法中，源数据库和目标数据库保持同步，但在将数据复制到目标数据库时，只有源数据库处理来自连接应用程序的事务。目标数据库在迁移期间不接受任何事务。

聚合函数

一种 SQL 函数，它对一组行进行操作并计算该组的单个返回值。聚合函数的示例包括 SUM 和 MAX。

AI

请参阅人工智能。

AIOps

请参阅人工智能运营。

匿名化

永久删除数据集中个人信息的过程。匿名化可以帮助保护个人隐私。匿名化数据不再被视为个人数据。

反模式

一种用于解决反复出现的问题的常用解决方案，而在这类问题中，此解决方案适得其反、无效或不如替代方案有效。

应用程序控制

一种安全方法，仅允许使用经批准的应用程序，以帮助保护系统免受恶意软件的侵害。

应用程序组合

有关组织使用的每个应用程序的详细信息的集合，包括构建和维护该应用程序的成本及其业务价值。这些信息是产品组合发现和分析过程的关键，有助于识别需要进行迁移、现代化和优化的应用程序并确定其优先级。

人工智能（AI）

计算机科学领域致力于使用计算技术执行通常与人类相关的认知功能，例如学习、解决问题和识别模式。有关更多信息，请参阅什么是人工智能？

人工智能操作 (AIOps)

使用机器学习技术解决运营问题、减少运营事故和人为干预以及提高服务质量的过程。有关如何在 AIOps AWS 迁移策略中使用的更多信息，请参阅操作集成指南。

非对称加密

一种加密算法，使用一对密钥，一个公钥用于加密，一个私钥用于解密。您可以共享公钥，因为它不用于解密，但对私钥的访问应受到严格限制。

原子性、一致性、隔离性、持久性（ACID）

一组软件属性，即使在出现错误、电源故障或其他问题的情况下，也能保证数据库的数据有效性和操作可靠性。

基于属性的访问权限控制（ABAC）

根据用户属性（如部门、工作角色和团队名称）创建精细访问权限的做法。有关更多信息，请参阅 AWS Identity and Access Management (I AM) 文档 AWS中的 AB AC。

权威数据来源

存储主要数据版本的位置，被认为是最可靠的信息源。您可以将数据从权威数据来源复制到其他位置，以便处理或修改数据，例如对数据进行匿名化、编辑或假名化。

可用区

中的一个不同位置 AWS 区域 ，不受其他可用区域故障的影响，并向同一区域中的其他可用区提供低成本、低延迟的网络连接。

AWS 云采用框架 (AWS CAF)

该框架包含指导方针和最佳实践 AWS ，可帮助组织制定高效且有效的计划，以成功迁移到云端。AWS CAF将指导分为六个重点领域，称为视角：业务、人员、治理、平台、安全和运营。业务、人员和治理角度侧重于业务技能和流程；平台、安全和运营角度侧重于技术技能和流程。例如，人

员角度针对的是负责人力资源（HR）、人员配置职能和人员管理的利益相关者。从这个角度来看，AWS CAF 为人员发展、培训和沟通提供了指导，以帮助组织为成功采用云做好准备。有关更多信息，请参阅 AWS CAF 网站和 AWS CAF 白皮书。

AWS 工作负载资格框架 (AWS WQF)

一种评估数据库迁移工作负载、推荐迁移策略和提供工作估算的工具。 AWS WQF 包含在 AWS Schema Conversion Tool (AWS SCT) 中。它用来分析数据库架构和代码对象、应用程序代码、依赖关系和性能特征，并提供评测报告。

B

恶意机器人

一种旨在扰乱或伤害个人或组织的机器人。

BCP

请参阅业务连续性计划。

行为图

一段时间内资源行为和交互的统一交互式视图。您可以使用 Amazon Detective 的行为图来检查失败的登录尝试、可疑的 API 调用和类似的操作。有关更多信息，请参阅 Detective 文档中的行为图中的数据。

大端序系统

一个先存储最高有效字节的系统。另请参阅字节顺序。

二进制分类

一种预测二进制结果（两个可能的类别之一）的过程。例如，您的 ML 模型可能需要预测诸如“该电子邮件是否为垃圾邮件？”或“这个产品是书还是汽车？”之类的问题

bloom 筛选条件

一种概率性、内存高效的数据结构，用于测试元素是否为集合的成员。

蓝/绿部署

一种部署策略，您可以创建两个独立但完全相同的环境。在一个环境中运行当前应用程序版本（蓝色），在另一个环境中运行新应用程序版本（绿色）。此策略可帮助您在影响最小的情况下快速回滚。

自动程序

一种通过互联网运行自动任务并模拟人类活动或交互的软件应用程序。有些机器人是有用或有益的，例如在互联网上索引信息的 Web 爬网程序。还有一些被称为恶意机器人的机器人，其目的是扰乱或伤害个人或组织。

僵尸网络

被恶意软件感染并受单方（称为僵尸网络控制者或僵尸网络操作者）控制的僵尸网络。僵尸网络是最著名的扩展机器人及其影响力的机制。

分支

代码存储库的一个包含区域。在存储库中创建的第一个分支是主分支。您可以从现有分支创建新分支，然后在新分支中开发功能或修复错误。为构建功能而创建的分支通常称为功能分支。当功能可以发布时，将功能分支合并回主分支。有关更多信息，请参阅关于分支（GitHub 文档）。

紧急（break-glass）访问

在特殊情况下，通过批准的流程，用户 AWS 账户 可以快速访问他们通常没有访问权限的内容。有关更多信息，请参阅 AWS Well-Architected Guidance 中的 Implement break-glass procedures 指示器。

棕地策略

您环境中的现有基础设施。在为系统架构采用棕地策略时，您需要围绕当前系统和基础设施的限制来设计架构。如果您正在扩展现有基础设施，则可以将棕地策略和全新策略混合。

缓冲区缓存

存储最常访问的数据的内存区域。

业务能力

企业如何创造价值（例如，销售、客户服务或营销）。微服务架构和开发决策可以由业务能力驱动。有关更多信息，请参阅在 AWS上运行容器化微服务白皮书中的围绕业务能力进行组织部分。

业务连续性计划（BCP）

一项计划，旨在应对大规模迁移等破坏性事件对运营的潜在影响，并使企业能够快速恢复运营。

C

CAF

请参阅 AWS 云采用框架。

金丝雀部署

缓慢而渐进地向最终用户发布版本。当您确信无误后，即可部署新版本，并完全替换当前版本。

CCoE

请参阅云卓越中心。

CDC

请参阅更改数据捕获。

更改数据捕获（CDC）

跟踪数据来源（如数据库表）的更改并记录有关更改的元数据的过程。您可以将 CDC 用于各种目的，例如审计或复制目标系统中的更改以保持同步。

混沌工程

故意引入故障或破坏性事件来测试系统的韧性。您可以使用 AWS Fault Injection Service (AWS FIS) 来执行实验，对您的 AWS 工作负载施加压力并评估其响应。

CI/CD

请参阅持续集成和持续交付。

分类

一种有助于生成预测的分类流程。分类问题的 ML 模型预测离散值。离散值始终彼此不同。例如，一个模型可能需要评估图像中是否有汽车。

客户端加密

在目标 AWS 服务 收到数据之前，对数据进行本地加密。

云卓越中心 (CCoE)

一个多学科团队，负责推动整个组织的云采用工作，包括开发云最佳实践、调动资源、制定迁移时间表、领导组织完成大规模转型。有关更多信息，请参阅 AWS Cloud 企业战略博客上的 CCoE 帖子。

云计算

通常用于远程数据存储和 IoT 设备管理的云技术。云计算通常连接到边缘计算技术。

云运营模型

在 IT 组织中，一种用于构建、完善和优化一个或多个云环境的运营模型。有关更多信息，请参阅构建您的云运营模型。

云采用阶段

组织迁移到 AWS Cloud中时通常会经历四个阶段：

- 项目 - 出于概念验证和学习目的，开展一些与云相关的项目
- 基础 — 进行基础投资以扩大云采用率（例如，创建着陆区、定义 CCo E、建立运营模型）
- 迁移 - 迁移单个应用程序
- 重塑 - 优化产品和服务，在云中创新

Stephen Orban在 AWS Cloud 企业战略博客的博客文章《云优先之旅和采用阶段》中定义了这些阶段。有关它们与 AWS 迁移策略的关系的信息，请参阅迁移准备指南。

CMDB

请参阅配置管理数据库。

代码存储库

通过版本控制过程存储和更新源代码和其他资产（如文档、示例和脚本）的位置。常见的云存储库包括 GitHub 或 Bitbucket Cloud。每个版本的代码都称为一个分支。在微服务结构中，每个存储库都专门用于一个功能。单个 CI/CD 管线可以使用多个存储库。

冷缓存

一种空的、填充不足或包含过时或不相关数据的缓冲区缓存。这会影响性能，因为数据库实例必须从主内存或磁盘读取，这比从缓冲区缓存读取要慢。

冷数据

很少访问的数据，且通常是历史数据。查询此类数据时，通常可以接受慢速查询。将这些数据转移到性能较低且成本更低的存储层或类别可以降低成本。

计算机视觉（CV）

一种 AI 领域，它使用机器学习来分析和提取数字图像和视频等视觉格式中的信息。例如，Amazon SageMaker AI 为 CV 提供了图像处理算法。

配置偏移

对于工作负载而言，一种偏离预期状态的配置更改。这可能会导致工作负载变得不合规，且通常是渐进的，不是故意的。

配置管理数据库（CMDB）

一种存储库，用于存储和管理有关数据库及其 IT 环境的信息，包括硬件和软件组件及其配置。您通常在迁移的产品组合发现和分析阶段使用来自 CMDB 的数据。

合规性包

一系列 AWS Config 规则和补救措施，您可以汇编这些规则和补救措施，以自定义您的合规性和安全性检查。您可以使用 YAML 模板将一致性包作为单个实体部署在 AWS 账户 和区域或整个组织中。有关更多信息，请参阅 AWS Config 文档中的一致性包。

持续集成和持续交付（CI/CD）

自动执行软件发布过程的源代码、构建、测试、暂存和生产阶段的过程。 CI/CD 通常被描述为管道。 CI/CD 可以帮助您实现流程自动化、提高生产力、提高代码质量和更快地交付。有关更多信息，请参阅持续交付的优势。CD 也可以表示持续部署。有关更多信息，请参阅持续交付与持续部署。

CV

请参阅计算机视觉。

D

静态数据

网络中静止的数据，例如存储中的数据。

数据分类

根据网络中数据的关键性和敏感性对其进行识别和分类的过程。它是任何网络安全风险管理策略的关键组成部分，因为它可以帮助您确定对数据的适当保护和保留控制。数据分类是 Well-Architecte AWS d Framework 中安全支柱的一个组成部分。有关详细信息，请参阅数据分类。

数据漂移

生产数据与用来训练机器学习模型的数据之间的有意义差异，或者输入数据随时间推移的有意义变化。数据漂移可能降低机器学习模型预测的整体质量、准确性和公平性。

传输中数据

在网络中主动移动的数据，例如在网络资源之间移动的数据。

数据网格

一种架构框架，可提供分布式、去中心化的数据所有权以及集中式管理和治理。

数据最少化

仅收集并处理绝对必要数据的原则。在中进行数据最小化 AWS Cloud 可以降低隐私风险、成本和分析碳足迹。

数据边界

AWS 环境中的一组预防性防护措施，可帮助确保只有可信身份才能访问来自预期网络的可信资源。有关更多信息，请参阅在上构建数据边界。 AWS

数据预处理

将原始数据转换为 ML 模型易于解析的格式。预处理数据可能意味着删除某些列或行，并处理缺失、不一致或重复的值。

数据溯源

在数据的整个生命周期跟踪其来源和历史的过程，例如数据如何生成、传输和存储。

数据主体

正在收集和处理其数据的个人。

数据仓库

一种支持商业智能（例如分析）的数据管理系统。数据仓库通常包含大量历史数据，通常用于查询和分析。

数据库定义语言（DDL）

在数据库中创建或修改表和对象结构的语句或命令。

数据库操作语言（DML）

在数据库中修改（插入、更新和删除）信息的语句或命令。

DDL

请参阅数据库定义语言。

深度融合

组合多个深度学习模型进行预测。您可以使用深度融合来获得更准确的预测或估算预测中的不确定性。

深度学习

一个 ML 子字段使用多层人工神经网络来识别输入数据和感兴趣的目标变量之间的映射。

defense-in-depth

一种信息安全方法，经过深思熟虑，在整个计算机网络中分层实施一系列安全机制和控制措施，以保护网络及其中数据的机密性、完整性和可用性。当你采用这种策略时 AWS，你会在 AWS

Organizations 结构的不同层面添加多个控件来帮助保护资源。例如，一种 defense-in-depth方法可以结合多因素身份验证、网络分段和加密。

委派管理员

在中 AWS Organizations，兼容的服务可以注册 AWS 成员帐户来管理组织的帐户并管理该服务的权限。此账户被称为该服务的委托管理员。有关更多信息和兼容服务列表，请参阅 AWS Organizations 文档中使用 AWS Organizations的服务。

部署

使应用程序、新功能或代码修复在目标环境中可用的过程。部署涉及在代码库中实现更改，然后在应用程序的环境中构建和运行该代码库。

开发环境

请参阅环境。

侦测性控制

一种安全控制，在事件发生后进行检测、记录日志和发出提醒。这些控制是第二道防线，提醒您注意绕过现有预防性控制的安全事件。有关更多信息，请参阅在 AWS上实施安全控制中的侦测性控制。

开发价值流映射（DVSM）

用于识别对软件开发生命周期中的速度和质量产生不利影响的限制因素并确定其优先级的流程。DVSM 扩展了最初为精益生产实践设计的价值流映射流程。其重点关注在软件开发过程中创造和转移价值所需的步骤和团队。

数字孪生

真实世界系统的虚拟再现，如建筑物、工厂、工业设备或生产线。数字孪生支持预测性维护、远程监控和生产优化。

维度表

星型架构中的一种较小的表，其中包含事实表中定量数据的数据属性。维度表属性通常是文本字段或行为类似于文本的离散数字。这些属性通常用于查询约束、筛选和结果集标注。

灾难

阻止工作负载或系统在其主要部署位置实现其业务目标的事件。这些事件可能是自然灾害、技术故障或人为操作的结果，例如无意的配置错误或恶意软件攻击。

灾难恢复（DR）

您用来最大程度地减少由灾难造成的停机时间和数据丢失的策略和流程。有关更多信息，请参阅 Well-Architected Fr ame AWS work 中的 “工作负载灾难恢复：云端 AWS 恢复”。

DML

请参阅数据库操作语言。

领域驱动设计

一种开发复杂软件系统的方法，通过将其组件连接到每个组件所服务的不断发展的领域或核心业务目标。Eric Evans 在其著作领域驱动设计：软件核心复杂性应对之道（Boston: Addison-Wesley Professional, 2003）中介绍了这一概念。有关如何将领域驱动设计与 strangler fig 模式结合使用的信息，请参阅使用容器和 Amazon API Gateway 逐步将原有的 Microsoft ASP.NET（ASMX）Web 服务现代化。

DR

请参阅灾难恢复。

偏差检测

跟踪与基准配置的偏差。例如，您可以使用 AWS CloudFormation 来检测系统资源中的偏差，也可以使用 AWS Control Tower 来检测着陆区中可能影响监管要求合规性的变化。

DVSM

请参阅开发价值流映射。

E

EDA

请参阅探索性数据分析。

EDI

请参阅电子数据交换。

边缘计算

该技术可提高位于 IoT 网络边缘的智能设备的计算能力。与云计算比较时，边缘计算可以减少通信延迟并缩短响应时间。

电子数据交换（EDI）

组织之间业务文件的自动交换。有关更多信息，请参阅什么是电子数据交换。

加密

一种将人类可读的纯文本数据转换为加密文字的计算流程。

加密密钥

由加密算法生成的随机位的加密字符串。密钥的长度可能有所不同，而且每个密钥都设计为不可预测且唯一。

字节顺序

字节在计算机内存中的存储顺序。大端序系统先存储最高有效字节。小端序系统先存储最低有效字节。

端点

请参阅服务端点。

端点服务

一种可以在虚拟私有云（VPC）中托管，与其他用户共享的服务。您可以使用其他 AWS 账户 或 AWS Identity and Access Management (IAM) 委托人创建终端节点服务， AWS PrivateLink 并向其授予权限。这些账户或主体可通过创建接口 VPC 端点来私密地连接到您的端点服务。有关更多信息，请参阅 Amazon Virtual Private Cloud（Amazon VPC）文档中的创建端点服务。

企业资源规划（ERP）

一种自动化和管理企业关键业务流程（例如会计、MES 和项目管理）的系统。

信封加密

用另一个加密密钥对加密密钥进行加密的过程。有关更多信息，请参阅 AWS Key Management Service (AWS KMS) 文档中的信封加密。

环境

正在运行的应用程序的实例。以下是云计算中常见的环境类型：

- 开发环境 — 正在运行的应用程序的实例，只有负责维护应用程序的核心团队才能使用。开发环境用于测试更改，然后再将其提升到上层环境。这类环境有时称为测试环境。
- 下层环境 — 应用程序的所有开发环境，比如用于初始构建和测试的环境。

- 生产环境 — 最终用户可以访问的正在运行的应用程序的实例。在 CI/CD 管道中，生产环境是最后一个部署环境。
- 上层环境 — 除核心开发团队以外的用户可以访问的所有环境。这可能包括生产环境、预生产环境和用户验收测试环境。

epic

在敏捷方法学中，有助于组织工作和确定优先级的功能类别。epics 提供了对需求和实施任务的总体描述。例如， AWS CAF 安全史诗包括身份和访问管理、侦探控制、基础设施安全、数据保护和事件响应。有关 AWS 迁移策略中 epics 的更多信息，请参阅计划实施指南。

ERP

请参阅企业资源规划。

探索性数据分析（EDA）

分析数据集以了解其主要特征的过程。您收集或汇总数据，并进行初步调查，以发现模式、检测异常并检查假定情况。EDA 通过计算汇总统计数据和创建数据可视化得以执行。

F

事实表

星型架构中的中心表。它存储有关业务运营的定量数据。通常，事实表包含两种类型的列：包含度量的列和包含维度表外键的列。

快速失效机制

一种使用频繁且增量式的测试来缩短开发生命周期的理念。这是敏捷方法的关键部分。

故障隔离边界

在中 AWS Cloud，诸如可用区 AWS 区域、控制平面或数据平面之类的边界，它限制了故障的影响并有助于提高工作负载的弹性。有关更多信息，请参阅 AWS 故障隔离边界。

功能分支

请参阅分支。

特征

您用来进行预测的输入数据。例如，在制造环境中，特征可能是定期从生产线捕获的图像。

特征重要性

特征对于模型预测的重要性。这通常表示为数值分数，可以通过各种技术进行计算，例如 Shapley 加法解释（SHAP）和积分梯度。有关更多信息，请参阅使用机器学习模型的可解释性 AWS。

功能转换

为 ML 流程优化数据，包括使用其他来源丰富数据、扩展值或从单个数据字段中提取多组信息。这使得 ML 模型能从数据中获益。例如，如果您将“2021-05-27 00:15:37”日期分解为“2021”、“五月”、“星期四”和“15”，则可以帮助学习与不同数据成分相关的算法学习精细模式。

少样本提示

在要求 LLM 执行类似任务之前，先向其提供少量示例，以演示任务和预期输出。此技术是上下文内学习的一种应用，其中模型可以从提示中嵌入的示例（样本）中学习。对于需要特定格式、推理或领域知识的任务，少样本提示可能非常有效。另请参阅零样本提示。

FGAC

请参阅精细访问控制。

精细访问控制（FGAC）

使用多个条件允许或拒绝访问请求。

快闪迁移

一种数据库迁移方法，通过更改数据捕获使用连续数据复制，在极短的时间内迁移数据，而非使用分阶段方法。目标是将停机时间降至最低。

FM

请参阅基础模型。

基础模型（FM）

一个大型深度学习神经网络，一直在广义和未标记数据的大量数据集上进行训练。 FMs 能够执行各种各样的一般任务，例如理解语言、生成文本和图像以及用自然语言进行对话。有关更多信息，请参阅什么是基础模型。

G

生成式人工智能

AI 模型的一个子集，这些模型已经过大量数据训练，可以使用简单的文本提示来创建新的内容和构件，例如图像、视频、文本和音频。有关更多信息，请参阅什么是生成式人工智能。

地理阻止

请参阅地理限制。

地理限制（地理阻止）

在 Amazon 中 CloudFront，一种阻止特定国家/地区的用户访问内容分发的选项。您可以使用允许列表或阻止列表来指定已批准和已禁止的国家/地区。有关更多信息，请参阅 CloudFront 文档中的限制内容的地理分布。

GitFlow 工作流程

一种方法，在这种方法中，下层和上层环境在源代码存储库中使用不同的分支。Gitflow 工作流程被认为是传统的工作流程，而基于中继的工作流程则是现代的、首选的方法。

黄金映像

系统或软件的快照，用作部署该系统或软件的新实例的模板。例如，在制造业中，黄金映像可用于在多个设备上预调配软件，并有助于提高设备制造操作的速度、可扩展性和生产效率。

全新策略

在新环境中缺少现有基础设施。在对系统架构采用全新策略时，您可以选择所有新技术，而不受对现有基础设施（也称为棕地）兼容性的限制。如果您正在扩展现有基础设施，则可以将棕地策略和全新策略混合。

防护机制

帮助管理各组织单位的资源、策略和合规性的高级规则 (OUs)。预防性防护机制会执行策略以确保符合合规性标准。它们是使用服务控制策略和 IAM 权限边界实现的。侦测性护栏会检测策略违规和合规性问题，并生成提醒以进行修复。它们通过使用 AWS Config、、Amazon、 AWS Security Hub CSPM GuardDuty AWS Trusted Advisor、Amazon Inspector 和自定义 AWS Lambda 支票来实现。

H

HA

请参阅高可用性。

异构数据库迁移

将源数据库迁移到使用不同数据库引擎的目标数据库（例如，从 Oracle 迁移到 Amazon Aurora）。异构迁移通常是重新架构工作的一部分，而转换架构可能是一项复杂的任务。AWS 提供了 AWS SCT 来帮助实现架构转换。

高可用性（HA）

在遇到挑战或灾难时，工作负载无需干预即可连续运行的能力。HA 系统旨在自动进行故障转移、持续提供良好性能，并以最小的性能影响处理不同负载和故障。

历史数据库现代化

一种用于实现运营技术（OT）系统现代化和升级以更好满足制造业需求的方法。历史数据库是一种用于收集和存储工厂中各种来源数据的数据库。

保留数据

从用于训练机器学习模型的数据集中保留的一部分标注的历史数据。通过将模型预测与保留数据进行比较，您可以使用保留数据来评估模型性能。

同构数据库迁移

将源数据库迁移到共享同一数据库引擎的目标数据库（例如，从 Microsoft SQL Server 迁移到 Amazon RDS for SQL Server）。同构迁移通常是更换主机或更换平台工作的一部分。您可以使用本机数据库实用程序来迁移架构。

热数据

经常访问的数据，例如实时数据或近期的转化数据。这些数据通常需要高性能存储层或存储类别才能提供快速的查询响应。

修补程序

针对生产环境中关键问题的紧急修复。由于其紧迫性，修补程序通常是在典型的 DevOps 发布工作流程之外进行的。

hypercare 周期

割接之后，迁移团队立即管理和监控云中迁移的应用程序以解决任何问题的时间段。通常，这个周期持续 1-4 天。在 hypercare 周期结束时，迁移团队通常会将应用程序的责任移交给云运营团队。

我

IaC

请参阅基础设施即代码。

基于身份的策略

附加到一个或多个 IAM 委托人的策略，用于定义他们在 AWS Cloud 环境中的权限。

空闲应用程序

90 天内平均 CPU 和内存使用率在 5% 到 20% 之间的应用程序。在迁移项目中，通常会停用这些应用程序或将其保留在本地。

IIoT

请参阅工业物联网。

不可变基础设施

一种模型，可为生产工作负载部署新的基础设施，而不是更新、修补或修改现有基础设施。不可变基础设施本质上比可变基础设施更一致、更可靠、更可预测。有关更多信息，请参阅 AWS Well-Architected Framework 中的使用不可变基础设施进行部署最佳实践。

入站（入口）VPC

在 AWS 多账户架构中，一种接受、检查和路由来自应用程序外部的网络连接的 VPC。AWS 安全参考架构建议设置您的网络帐户，包括入站、出站和检查， VPCs 以保护您的应用程序与更广泛的互联网之间的双向接口。

增量迁移

一种割接策略，在这种策略中，您可以将应用程序分成小部分进行迁移，而不是一次性完整割接。例如，您最初可能只将几个微服务或用户迁移到新系统。在确认一切正常后，您可以逐步迁移其他微服务或用户，直到停用遗留系统。这种策略降低了大规模迁移带来的风险。

工业 4.0

该术语由 Klaus Schwab 在 2016 年提出，指的是通过连接、实时数据、自动化、分析和 AI/ML 的进步来实现制造流程的现代化。

基础设施

应用程序环境中包含的所有资源和资产。

基础设施即代码（IaC）

通过一组配置文件预调配和管理应用程序基础设施的过程。IaC 旨在帮助您集中管理基础设施、实现资源标准化和快速扩展，使新环境具有可重复性、可靠性和一致性。

工业物联网 (IIoT)

在工业领域使用联网的传感器和设备，例如制造业、能源、汽车、医疗保健、生命科学和农业。有关更多信息，请参阅制定工业物联网 (IIoT) 数字化转型战略。

检查 VPC

在 AWS 多账户架构中，一种集中式 VPC，用于管理对 VPCs（相同或不同 AWS 区域）、互联网和本地网络之间的网络流量的检查。AWS 安全参考架构建议设置您的网络帐户，包括入站、出站和检查， VPCs 以保护您的应用程序与更广泛的互联网之间的双向接口。

物联网（IoT）

由带有嵌入式传感器或处理器的连接物理对象组成的网络，这些传感器或处理器通过互联网或本地通信网络与其他设备和系统进行通信。有关更多信息，请参阅什么是 IoT？

可解释性

它是机器学习模型的一种特征，描述了人类可以理解模型的预测如何取决于其输入的程度。有关更多信息，请参阅使用机器学习模型的可解释性 AWS。

物联网

请参阅物联网。

IT 信息库（ITIL）

提供 IT 服务并使这些服务符合业务要求的一套最佳实践。ITIL 是 ITSM 的基础。

IT 服务管理（ITSM）

为组织设计、实施、管理和支持 IT 服务的相关活动。有关将云运营与 ITSM 工具集成的信息，请参阅运营集成指南。

ITIL

请参阅 IT 信息库。

ITSM

请参阅 IT 服务管理。

L

基于标签的访问控制（LBAC）

强制访问控制（MAC）的一种实施方式，其中明确为用户和数据本身分配了安全标签值。用户安全标签和数据安全标签之间的交集决定了用户可以看到哪些行和列。

登录区

landing zone 是一个架构精良的多账户 AWS 环境，具有可扩展性和安全性。这是一个起点，您的组织可以从这里放心地在安全和基础设施环境中快速启动和部署工作负载和应用程序。有关登录区的更多信息，请参阅设置安全且可扩展的多账户 AWS 环境。

大语言模型（LLM）

一种基于大量数据进行预训练的深度学习 AI 模型。LLM 可以执行多项任务，例如回答问题、总结文档、将文本翻译成其他语言以及完成句子。有关更多信息，请参阅什么是 LLMs。

大规模迁移

迁移 300 台或更多服务器。

LBAC

请参阅基于标签的访问控制。

最低权限

授予执行任务所需的最低权限的最佳安全实践。有关更多信息，请参阅 IAM 文档中的应用最低权限许可。

直接迁移

请参阅 7 R。

小端序系统

一个先存储最低有效字节的系统。另请参阅字节顺序。

LLM

请参阅大型语言模型。

下层环境

请参阅环境。

M

机器学习（ML）

一种使用算法和技术进行模式识别和学习的人工智能。ML 对记录的数据（例如物联网（IoT）数据）进行分析和学习，以生成基于模式的统计模型。有关更多信息，请参阅机器学习。

主分支

请参阅分支。

恶意软件

旨在危害计算机安全或隐私的软件。恶意软件可能会破坏计算机系统、泄露敏感信息或获得未经授权的访问权限。恶意软件的示例包括病毒、蠕虫、勒索软件、木马、间谍软件和键盘记录器。

托管式服务

AWS 服务 它 AWS 运行基础设施层、操作系统和平台，您可以访问端点来存储和检索数据。Amazon Simple Storage Service（Amazon S3）和 Amazon DynamoDB 就是托管服务的示例。这些服务也称为抽象服务。

制造执行系统（MES）

一种软件系统，用于跟踪、监控、记录和控制在车间将原材料转化为成品的生产过程。

MAP

请参阅迁移加速计划。

机制

一个完整的过程，您可以在其中创建工具，推动工具的采用，然后检查结果以进行调整。机制是一种在运作过程中自我强化和改善的循环。有关更多信息，请参阅在 Well-Architect AWS ed 框架中构建机制。

成员账户

AWS 账户 除属于组织中的管理账户之外的所有账户 AWS Organizations。一个账户一次只能是一个组织的成员。

MES

请参阅制造执行系统。

消息队列遥测传输（MQTT）

一种基于发布/订阅模式的轻量级 machine-to-machine (M2M) 通信协议，适用于资源受限的物联网设备。

微服务

一种小型的独立服务，通过明确的定义进行通信 APIs ，通常由小型的独立团队拥有。例如，保险系统可能包括映射到业务能力（如销售或营销）或子域（如购买、理赔或分析）的微服务。微服务

的好处包括敏捷、灵活扩展、易于部署、可重复使用的代码和恢复能力。有关更多信息，请参阅使用 AWS 无服务器服务集成微服务。

微服务架构

一种使用独立组件构建应用程序的方法，这些组件将每个应用程序进程作为微服务运行。这些微服务使用轻量级通过定义明确的接口进行通信。 APIs该架构中的每个微服务都可以更新、部署和扩展，以满足对应用程序特定功能的需求。有关更多信息，请参阅在上实现微服务。 AWS

迁移加速计划（MAP）

AWS 该计划提供咨询支持、培训和服务，以帮助组织为迁移到云奠定坚实的运营基础，并帮助抵消迁移的初始成本。MAP 提供了一种以系统的方式执行遗留迁移的迁移方法，以及一套用于自动执行和加速常见迁移场景的工具。

大规模迁移

将大部分应用程序组合分波迁移到云中的过程，在每一波中以更快的速度迁移更多应用程序。本阶段使用从早期阶段获得的最佳实践和经验教训，实施由团队、工具和流程组成的迁移工厂，通过自动化和敏捷交付简化工作负载的迁移。这是 AWS 迁移策略的第三阶段。

迁移工厂

跨职能团队，通过自动化、敏捷的方法简化工作负载迁移。迁移工厂团队通常包括运营、业务分析师和所有者、迁移工程师、开发人员和冲刺 DevOps 领域的专业人员。20% 到 50% 的企业应用程序组合由可通过工厂方法优化的重复模式组成。有关更多信息，请参阅本内容集中有关迁移工厂的讨论和云迁移工厂指南。

迁移元数据

有关完成迁移所需的应用程序和服务器的信息。每种迁移模式都需要一套不同的迁移元数据。迁移元数据的示例包括目标子网、安全组和 AWS 账户。

迁移模式

一种可重复的迁移任务，详细列出了迁移策略、迁移目标以及所使用的迁移应用程序或服务。示例：使用 AWS 应用程序迁移服务重新托管向 Amazon EC2 的迁移。

迁移组合评测（MPA）

一种在线工具，提供了用于验证迁移到 AWS Cloud的业务案例的信息。MPA 提供了详细的组合评测（服务器规模调整、定价、TCO 比较、迁移成本分析）以及迁移计划（应用程序数据分析和数据收集、应用程序分组、迁移优先级排序和波次规划）。所有 AWS 顾问和 APN 合作伙伴顾问均可免费使用 MPA 工具（需要登录）。

迁移准备情况评测（MRA）

使用 AWS CAF 深入了解组织的云就绪状态、确定优势和劣势以及制定行动计划以缩小已发现差距的过程。有关更多信息，请参阅迁移准备指南。MRA 是 AWS 迁移策略的第一阶段。

迁移策略

将工作负载迁移到 AWS Cloud的方法。有关更多信息，请参见术语表中的 7 R 词条，以及动员您的组织以加快大规模迁移。

ML

请参阅机器学习。

现代化

将过时的（原有的或单体）应用程序及其基础设施转变为云中敏捷、弹性和高度可用的系统，以降低成本、提高效率和利用创新。有关更多信息，请参阅在 AWS Cloud中实现应用程序现代化的策略。

现代化准备情况评估

一种评估方式，有助于确定组织应用程序的现代化准备情况；确定收益、风险和依赖关系；确定组织能够在多大程度上支持这些应用程序的未来状态。评估结果是目标架构的蓝图、详细说明现代化进程发展阶段和里程碑的路线图以及解决已发现差距的行动计划。有关更多信息，请参阅在 AWS Cloud中评估应用程序的现代化准备情况。

单体应用程序（单体式）

作为具有紧密耦合进程的单个服务运行的应用程序。单体应用程序有几个缺点。如果某个应用程序功能的需求激增，则必须扩展整个架构。随着代码库的增长，添加或改进单体应用程序的功能也会变得更加复杂。若要解决这些问题，可以使用微服务架构。有关更多信息，请参阅将单体分解为微服务。

MPA

请参阅迁移组合评测。

MQTT

请参阅消息队列遥测传输。

多分类器

一种帮助为多个类别生成预测（预测两个以上结果之一）的过程。例如，ML 模型可能会询问“这个产品是书、汽车还是手机？”或“此客户最感兴趣什么类别的产品？”

可变基础设施

一种用于更新和修改生产工作负载的现有基础设施的模型。为了提高一致性、可靠性和可预测性，Well-Architect AWS ed Framework 建议使用不可变基础设施作为最佳实践。

O

OAC

请参阅来源访问控制。

OAI

请参阅来源访问身份。

OCM

请参阅组织变革管理。

离线迁移

一种迁移方法，在这种方法中，源工作负载会在迁移过程中停止运行。这种方法会延长停机时间，通常用于小型非关键工作负载。

OI

请参阅运营集成。

OLA

请参阅运营级别协议。

在线迁移

一种迁移方法，在这种方法中，源工作负载无需离线即可复制到目标系统。在迁移过程中，连接工作负载的应用程序可以继续运行。这种方法的停机时间为零或最短，通常用于关键生产工作负载。

OPC-UA

请参阅开放流程通信 – 统一架构。

开放流程通信 – 统一架构（OPC-UA）

一种用于工业自动化的 machine-to-machine（M2M）通信协议。OPC-UA 提供了一个包含数据加密、身份验证和授权方案的互操作性标准。

运营级别协议（OLA）

一项协议，阐明了 IT 职能部门承诺相互交付的内容，以支持服务水平协议（SLA）。

运营准备情况审查（ORR）

一份问题核对清单和关联的最佳实践，可帮助您了解、评估、预防或缩小事件和可能的故障的范围。有关更多信息，请参阅 AWS Well-Architected Framework 中的运营准备情况审查（ORR）。

运营技术（OT）

与物理环境配合使用以控制工业运营、设备和基础设施的硬件和软件系统。在制造业中，OT 和信息技术（IT）系统的集成是工业 4.0 转型的关键重点。

运营整合（OI）

在云中实现运营现代化的过程，包括就绪计划、自动化和集成。有关更多信息，请参阅运营整合指南。

组织跟踪

由 AWS CloudTrail 此创建的跟踪记录组织 AWS 账户 中所有人的所有事件 AWS Organizations。该跟踪是在每个 AWS 账户 中创建的，属于组织的一部分，并跟踪每个账户的活动。有关更多信息，请参阅 CloudTrail文档中的为组织创建跟踪。

组织变革管理（OCM）

一个从人员、文化和领导力角度管理重大、颠覆性业务转型的框架。OCM 通过加快变革采用、解决过渡问题以及推动文化和组织变革，帮助组织为新系统和战略做好准备和过渡。在 AWS 迁移策略中，该框架被称为人员加速，因为云采用项目需要变更的速度。有关更多信息，请参阅 OCM 指南。

来源访问控制（OAC）

在中 CloudFront，一个增强的选项，用于限制访问以保护您的亚马逊简单存储服务 (Amazon S3) 内容。OAC 全部支持所有 S3 存储桶 AWS 区域、使用 AWS KMS (SSE-KMS) 进行服务器端加密，以及对 S3 存储桶的动态PUT和DELETE请求。

来源访问身份（OAI）

在中 CloudFront，一个用于限制访问权限以保护您的 Amazon S3 内容的选项。当您使用 OAI 时，CloudFront 会创建一个 Amazon S3 可以对其进行身份验证的委托人。经过身份验证的委托人只能通过特定 CloudFront 分配访问 S3 存储桶中的内容。另请参阅 OAC，其中提供了更精细和增强的访问控制。

ORR

请参阅运营准备情况审查。

OT

请参阅运营技术。

出站（出口）VPC

在 AWS 多账户架构中，一种处理从应用程序内部启动的网络连接的 VPC。AWS 安全参考架构建议设置您的网络帐户，包括入站、出站和检查， VPCs 以保护您的应用程序与更广泛的互联网之间的双向接口。

P

权限边界

附加到 IAM 主体的 IAM 管理策略，用于设置用户或角色可以拥有的最大权限。有关更多信息，请参阅 IAM 文档中的权限边界。

个人身份信息（PII）

直接查看其他相关数据或与之配对时可用于合理推断个人身份的信息。PII 的示例包括姓名、地址和联系信息。

PII

请参阅个人身份信息。

playbook

一套预定义的步骤，用于捕获与迁移相关的工作，例如在云中交付核心运营功能。playbook 可以采用脚本、自动化运行手册的形式，也可以是操作现代化环境所需的流程或步骤的摘要。

PLC

请参阅可编程逻辑控制器。

PLM

请参阅产品生命周期管理。

policy

一个对象，可以定义权限（请参阅基于身份的策略）、指定访问条件（请参阅基于资源的策略）或定义 AWS Organizations 的组织中所有账户的最大权限（请参阅服务控制策略）。

多语言持久性

根据数据访问模式和其他要求，独立选择微服务的数据存储技术。如果您的微服务采用相同的数据存储技术，它们可能会遇到实现难题或性能不佳。如果微服务使用最适合其需求的数据存储，则可以更轻松地实现微服务，并获得更好的性能和可扩展性。

组合评测

一个发现、分析和确定应用程序组合优先级以规划迁移的过程。有关更多信息，请参阅评估迁移准备情况。

谓词

返回 `true` 或 `false` 的查询条件，通常位于 `WHERE` 子句中。

谓词下推

一种数据库查询优化技术，可在传输之前筛选查询中的数据。这将减少从关系数据库检索和处理的数据量，并提高查询性能。

预防性控制

一种安全控制，旨在防止事件发生。这些控制是第一道防线，帮助防止未经授权的访问或对网络的意外更改。有关更多信息，请参阅在 AWS上实施安全控制中的预防性控制。

主体

中 AWS 可以执行操作和访问资源的实体。此实体通常是 IAM 角色的根用户或用户。 AWS 账户有关更多信息，请参阅 IAM 文档中角色术语和概念中的主体。

隐私设计

一种在整个开发过程中都考虑隐私的系统工程方法。

私有托管区

一个容器，其中包含有关您希望 Amazon Route 53 如何响应针对一个或多个 VPCs域名及其子域名的 DNS 查询的信息。有关更多信息，请参阅 Route 53 文档中的私有托管区的使用。

主动控制

一种安全控制，旨在防止部署不合规资源。这些控制会在资源预置之前对其进行扫描。如果资源与控制不兼容，则不会预置它。有关更多信息，请参阅 AWS Control Tower 文档中的控制参考指南，并参见在上实施安全控制中的主动控制 AWS。

产品生命周期管理（PLM）

对产品在其整个生命周期内的数据和流程的管理，从设计、开发和发布，到增长和成熟，再到衰退和淘汰。

生产环境

请参阅环境。

可编程逻辑控制器（PLC）

在制造业中，一种高度可靠、适应性强的计算机，用于监控机器并实现制造过程自动化。

提示串接

使用一个 LLM 提示的输出作为下一个提示的输入，以生成更好的响应。该技术用于将复杂的任务分解为子任务，或者迭代地完善或扩展初步响应。它有助于提高模型响应的准确性和相关性，并允许获得更精细的个性化结果。

假名化

用占位符值替换数据集中个人标识符的过程。假名化可以帮助保护个人隐私。假名化数据仍被视为个人数据。

publish/subscribe (pub/sub)

一种支持微服务间异步通信的模式，可提高可扩展性和响应能力。例如，在基于微服务的 MES 中，微服务可以将事件消息发布到其他微服务可以订阅的频道。系统可以在不更改发布服务的情况下添加新的微服务。

Q

查询计划

一系列用于访问 SQL 关系数据库系统中的数据的步骤，类似于指令。

查询计划回归

当数据库服务优化程序选择的最佳计划不如数据库环境发生特定变化之前时。这可能是由统计数据、约束、环境设置、查询参数绑定更改和数据库引擎更新造成的。

R

RACI 矩阵

请参阅责任、问责、咨询和知情（RACI）。

RAG

请参阅检索增强生成。

勒索软件

一种恶意软件，旨在阻止对计算机系统或数据的访问，直到付款为止。

RASCI 矩阵

请参阅责任、问责、咨询和知情（RACI）。

RCAC

请参阅行列访问控制。

只读副本

用于只读目的的数据库副本。您可以将查询路由到只读副本，以减轻主数据库的负载。

重新架构

请参阅 7 R。

恢复点目标（RPO）

自上一个数据恢复点以来可接受的最长时间。这决定了从上一个恢复点到服务中断之间可接受的数据丢失情况。

恢复时间目标（RTO）

服务中断和服务恢复之间可接受的最大延迟。

重构

请参阅 7 R。

Region

地理区域内的 AWS 资源集合。每一个 AWS 区域 都相互隔离，相互独立，以提供容错、稳定性和弹性。有关更多信息，请参阅指定您的账户可以使用的 AWS 区域。

回归

一种预测数值的 ML 技术。例如，要解决“这套房子的售价是多少？”的问题 ML 模型可以使用线性回归模型，根据房屋的已知事实（如建筑面积）来预测房屋的销售价格。

重新托管

请参阅 7 R。

版本

在部署过程中，推动生产环境变更的行为。

重新放置

请参阅 7 R。

更换平台

请参阅 7 R。

重新购买

请参阅 7 R。

韧性

应用程序抵御中断或从中断中恢复的能力。在 AWS Cloud中规划韧性时，高可用性和灾难恢复是常见的考虑因素。有关更多信息，请参阅 AWS Cloud 韧性。

基于资源的策略

一种附加到资源的策略，例如 AmazonS3 存储桶、端点或加密密钥。此类策略指定了允许哪些主体访问、支持的操作以及必须满足的任何其他条件。

责任、问责、咨询和知情（RACI）矩阵

定义参与迁移活动和云运营的所有各方的角色和责任的矩阵。矩阵名称源自矩阵中定义的责任类型：负责（R）、问责（A）、咨询（C）和知情（I）。支持（S）类型是可选的。如果包括支持，则该矩阵称为 RASCI 矩阵，如果将其排除在外，则称为 RACI 矩阵。

响应性控制

一种安全控制，旨在推动对不良事件或偏离安全基线的情况进行修复。有关更多信息，请参阅在AWS上实施安全控制中的响应性控制。

保留

请参阅 7 R。

停用

请参阅 7 R。

检索增强生成（RAG）

一种生成式人工智能技术，其中 LLM 在生成响应之前引用其训练数据来源之外的权威数据来源。例如，RAG 模型可以对组织的知识库或自定义数据执行语义搜索。有关更多信息，请参阅什么是 RAG。

轮换

定期更新密钥以使攻击者更难访问凭证的过程。

行列访问控制（RCAC）

使用已定义访问规则的基本、灵活的 SQL 表达式。RCAC 由行权限和列掩码组成。

RPO

请参阅恢复点目标。

RTO

请参阅恢复时间目标。

运行手册

执行特定任务所需的一套手动或自动程序。它们通常是为了简化重复性操作或高错误率的程序而设计的。

S

SAML 2.0

许多身份提供商 (IdPs) 使用的开放标准。此功能支持联合单点登录 (SSO)，因此用户无需在 IAM 中为组织中的所有人创建用户即可登录 AWS 管理控制台 或调用 AWS API 操作。有关基于 SAML 2.0 的联合身份验证的更多信息，请参阅 IAM 文档中的关于基于 SAML 2.0 的联合身份验证。

SCADA

请参阅监督控制和数据采集。

SCP

请参阅服务控制策略。

机密密钥

在中 AWS Secrets Manager，您以加密形式存储的机密或受限信息，例如密码或用户凭证。它由密钥值及其元数据组成。密钥值可以是二进制、单个字符串或多个字符串。有关更多信息，请参阅 Secrets Manager 文档中的什么是 Amazon Secrets Manager 密钥？。

安全设计

一种在整个开发过程中都考虑安全的系统工程方法。

安全控制

一种技术或管理防护机制，可防止、检测或降低威胁行为体利用安全漏洞的能力。安全控制有以下四种类型：预防性、检测性、响应性和主动性。

安全固化

缩小攻击面，使其更能抵御攻击的过程。这可能包括删除不再需要的资源、实施授予最低权限的最佳安全实践或停用配置文件中不必要的功能等操作。

安全信息和事件管理（SIEM）系统

结合了安全信息管理（SIM）和安全事件管理（SEM）系统的工具和服务。SIEM 系统会收集、监控和分析来自服务器、网络、设备和其他来源的数据，以检测威胁和安全漏洞，并生成警报。

安全响应自动化

一种预定义的程序化操作，旨在自动响应或修复安全事件。这些自动化可作为侦探或响应式安全控制措施，帮助您实施 AWS 安全最佳实践。自动响应操作的示例包括修改 VPC 安全组、修补 Amazon EC2 实例或轮换凭证。

服务器端加密

由接收数据的人在目的地对数据 AWS 服务 进行加密。

服务控制策略（SCP）

一种策略，用于集中控制组织中所有账户的权限 AWS Organizations。 SCPs 定义防护措施或限制管理员可以委托给用户或角色的操作。您可以使用 SCPs 允许列表或拒绝列表来指定允许或禁止哪些服务或操作。有关更多信息，请参阅 AWS Organizations 文档中的服务控制策略。

服务端点

的入口点的 URL AWS 服务。您可以使用端点，通过编程方式连接到目标服务。有关更多信息，请参阅 AWS 一般参考 中的 AWS 服务 端点。

服务水平协议（SLA）

一份协议，阐明了 IT 团队承诺向客户交付的内容，比如服务正常运行时间和性能。

服务水平指示器（SLI）

对服务性能方面的衡量，例如错误率、可用性或吞吐量。

服务水平目标（SLO）

代表服务运行状况的目标指标，由服务水平指示器衡量。

责任共担模式

描述您在云安全与合规方面共同承担 AWS 的责任的模型。 AWS 负责云的安全，而您则负责云中的安全。有关更多信息，请参阅责任共担模式。

SIEM

请参阅安全信息和事件管理系统。

单点故障（SPOF）

应用程序的单个关键组件出现故障，可能会中断系统。

SLA

请参阅服务水平协议。

SLI

请参阅服务水平指示器。

SLO

请参阅服务水平目标。

split-and-seed 模型

一种扩展和加速现代化项目的模式。随着新功能和产品发布的定义，核心团队会拆分以创建新的产品团队。这有助于扩展组织的能力和服务，提高开发人员的工作效率，支持快速创新。有关更多信息，请参阅在 AWS Cloud中实现应用程序现代化的分阶段方法。

SPOF

请参阅单点故障。

星型架构

一种数据库组织结构，它使用一个大型事实表来存储事务数据或测量数据，并使用一个或多个较小的维度表来存储数据属性。此结构专为在数据仓库中使用或用于商业智能目的而设计。

strangler fig 模式

一种通过逐步重写和替换系统功能直至可以停用原有的系统来实现单体系统现代化的方法。这种模式用无花果藤作为类比，这种藤蔓成长为一棵树，最终战胜并取代了宿主。该模式是由 Martin Fowler 提出的，作为重写单体系统时管理风险的一种方法。有关如何应用此模式的示例，请参阅使用容器和 Amazon API Gateway 逐步将原有的 Microsoft ASP.NET（ASMX）Web 服务现代化。

子网

您的 VPC 内的一个 IP 地址范围。子网必须位于单个可用区中。

监督控制和数据采集（SCADA）

在制造业中，一种使用硬件和软件来监控实物资产和生产操作的系统。

对称加密

一种加密算法，它使用相同的密钥来加密和解密数据。

综合测试

以模拟用户交互的方式测试系统，以检测潜在问题或监控性能。你可以使用 Amazon S CloudWatch ynthetics 来创建这些测试。

系统提示

一种为 LLM 提供上下文、说明或准则以指导其行为的技术。系统提示有助于设置上下文并制定与用户交互的规则。

T

标签

键值对，用作组织资源的元数据。 AWS 标签有助于您管理、识别、组织、搜索和筛选 资源。有关更多信息，请参阅标记您的 AWS 资源。

目标变量

您在监督式 ML 中尝试预测的值。这也被称为结果变量。例如，在制造环境中，目标变量可能是产品缺陷。

任务列表

一种通过运行手册用于跟踪进度的工具。任务列表包含运行手册的概述和要完成的常规任务列表。对于每项常规任务，它包括预计所需时间、所有者和进度。

测试环境

请参阅环境。

训练

为您的 ML 模型提供学习数据。训练数据必须包含正确答案。学习算法在训练数据中查找将输入数据属性映射到目标（您希望预测的答案）的模式。然后输出捕获这些模式的 ML 模型。然后，您可以使用 ML 模型对不知道目标的新数据进行预测。

中转网关

一个网络传输中心，可用于将您的网络 VPCs 和本地网络互连。有关更多信息，请参阅 AWS Transit Gateway 文档中的什么是公交网关。

基于中继的工作流程

一种方法，开发人员在功能分支中本地构建和测试功能，然后将这些更改合并到主分支中。然后，按顺序将主分支构建到开发、预生产和生产环境。

可信访问权限

向您指定的服务授予权限，该服务可代表您在其账户中执行任务。 AWS Organizations 当需要服务相关的角色时，受信任的服务会在每个账户中创建一个角色，为您执行管理任务。有关更多信息，请参阅 AWS Organizations 文档中的AWS Organizations 与其他 AWS 服务一起使用。

优化

更改训练过程的各个方面，以提高 ML 模型的准确性。例如，您可以通过生成标签集、添加标签，并在不同的设置下多次重复这些步骤来优化模型，从而训练 ML 模型。

双披萨团队

一个小 DevOps 团队，你可以用两个披萨来喂食。双披萨团队的规模可确保在软件开发过程中充分协作。

U

不确定性

这一概念指的是不精确、不完整或未知的信息，这些信息可能会破坏预测式 ML 模型的可靠性。不确定性有两种类型：认知不确定性是由有限的、不完整的数据造成的，而偶然不确定性是由数据中固有的噪声和随机性导致的。有关更多信息，请参阅量化深度学习系统中的不确定性指南。

无差别任务

也称为繁重工作，即创建和运行应用程序所必需的工作，但不能为最终用户提供直接价值或竞争优势。无差别任务的示例包括采购、维护和容量规划。

上层环境

请参阅环境。

V

vacuum 操作

一种数据库维护操作，包括在增量更新后进行清理，以回收存储空间并提高性能。

版本控制

跟踪更改的过程和工具，例如存储库中源代码的更改。

VPC 对等连接

两者之间的连接 VPCs ，允许您使用私有 IP 地址路由流量。有关更多信息，请参阅 Amazon VPC 文档中的什么是 VPC 对等连接。

漏洞

损害系统安全的软件缺陷或硬件缺陷。

W

热缓存

一种包含经常访问的当前相关数据的缓冲区缓存。数据库实例可以从缓冲区缓存读取，这比从主内存或磁盘读取要快。

暖数据

不常访问的数据。查询此类数据时，通常可以接受中速查询。

窗口函数

一种对与当前记录有某种关联的一组行执行计算的 SQL 函数。窗口函数对于处理任务很有用，例如计算移动平均值或根据当前行的相对位置访问行的值。

工作负载

一系列资源和代码，它们可以提供商业价值，如面向客户的应用程序或后端过程。

工作流

迁移项目中负责一组特定任务的职能小组。每个工作流都是独立的，但支持项目中的其他工作流。例如，组合工作流负责确定应用程序的优先级、波次规划和收集迁移元数据。组合工作流将这些资产交付给迁移工作流，然后迁移服务器和应用程序。

WORM

请参阅一次写入多次读取。

WQF

请参阅 AWS 工作负载资格鉴定框架。

一次写入多次读取（WORM）

一种存储模型，可一次写入数据并防止数据被删除或修改。授权用户可以根据需要多次读取数据，但无法对其进行更改。此数据存储基础设施被认为不可变。

Z

零日漏洞利用

一种利用零日漏洞的攻击，通常为恶意软件。

零日漏洞

生产系统中不可避免的缺陷或漏洞。威胁主体可能利用这种类型的漏洞攻击系统。开发人员经常因攻击而意识到该漏洞。

零样本提示

为 LLM 提供执行任务的说明，但没有可以帮助指导的示例（样本）。LLM 必须使用预先训练的知识来处理任务。零样本提示的有效性取决于任务的复杂性和提示的质量。另请参阅少样本提示。

僵尸应用程序

平均 CPU 和内存使用率低于 5% 的应用程序。在迁移项目中，通常会停用这些应用程序。

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。