雲端設計模式、架構和實作

AWS 方案指引

AWS 方案指引: 雲端設計模式、架構和實作

Table of Contents

雲端設計模式、架構和實作

Amazon Web Services (AWS)，Anitha Deenadayalan

2024 年 5 月 (文件歷史記錄)

本指南提供使用 AWS 服務實作常用現代化設計模式的指引。越來越多現代應用程式是透過使用微服務架構來達成可擴展性、改善發行速度、減少變更的影響範圍，以及減少迴歸的設計。如此可改善開發人員的生產力並提高靈活度、獲得更出色的創新能力，並更能專注在業務需求上。微服務架構還支援為服務和資料庫使用最佳技術，並提倡多語言程式碼和多語言持續性。

傳統上，單一應用程式會在單一程序中執行、使用一個資料存放區，以及在垂直擴展的伺服器上執行。相較之下，現代微服務應用程式是精細的、具有獨立的故障網域、在整個網路上作為服務執行，並且可以根據使用案例使用多個資料存放區。服務水平擴展，而單一交易可能跨越多個資料庫。開發團隊在使用微服務架構開發應用程式時，必須專注於網路通訊、多語言持續性、水平擴展、最終一致性以及跨資料存放區的交易處理。因此，現代化模式對於解決現代應用程式開發中常見的問題而言相當重要，且有助於加速軟體交付。

本指南為雲端架構師、技術主管、應用程式和企業擁有者，以及希望根據架構良好的最佳實務為設計模式選擇合適雲端架構的開發人員提供技術參考資料。本指南中討論的每個模式都針對微服務架構中的一或多個已知案例進行解決。本指南討論與每個模式相關的問題和考量事項、提供高階架構實作，並說明該模式的 AWS 實作。開放原始碼 GitHub 範例和研討會連結會在提供時提供。

本指南涵蓋下列模式：

- 防損毀層
- API 路由模式：
  - 主機名稱路由
  - 路徑路由
  - http 標頭路由
- 斷路器
- 事件來源
- 六角形架構
- 發布-訂閱
- 使用輪詢重試
- 系列事件模式：

  - 系列事件編排
  - 系列事件協同運作
- 散佈集合
- Strangler Fig
- 交易寄件匣

目標業務成果

透過使用本指南中討論的模式將應用程式現代化，您可以：

- 設計並實作可靠、安全且具有營運效率的架構，並針對成本和效能進行最佳化。
- 縮短需要這些模式的使用案例週期時間，以便您可以專注於組織特定的挑戰。
- 使用 AWS 服務標準化模式實作來加速開發。
- 協助您的開發人員建置現代應用程式，而不會有老舊技術的限制。

反貪汙層模式

意圖

反損毀層 (ACL) 模式可做為中介層，將網域模型語意從一個系統轉譯為另一個系統。它會先將上游邊界內容 (monolith) 的模型轉換為適合下游邊界內容 (microservice) 的模型，再使用上游團隊建立的通訊合約。當下游邊界內容包含核心子網域，或上游模型是無法修改的舊版系統時，此模式可能適用。當來電者的呼叫必須以透明方式重新導向至目標系統時，它還可以防止對來電者進行變更，從而降低轉換風險和業務中斷。

動機

在遷移過程中，當單一應用程式遷移至微服務時，新遷移服務的網域模型語意可能會有變更。當需要整體內的功能來呼叫這些微服務時，呼叫應該路由到遷移的服務，而不需要對呼叫服務進行任何變更。ACL 模式允許整體以透明的方式呼叫微服務，方法是做為轉接器或外觀圖層，將呼叫轉換為較新的語意。

適用性

考慮在下列情況下使用此模式：

- 您現有的單體應用程式必須與已遷移至微服務的函數通訊，而遷移的服務網域模型和語意與原始功能不同。
- 兩個系統具有不同的語意，需要交換資料，但修改一個系統以與其他系統相容並不實際。
- 您想要使用快速且簡化的方法，將一個系統調整為另一個系統，並將影響降至最低。
- 您的應用程式正在與外部系統通訊。

問題和考量

- 團隊相依性：當系統中的不同服務由不同的團隊擁有時，遷移服務中的新網域模型語意可能會導致呼叫系統中的變更。不過，團隊可能無法以協調的方式進行這些變更，因為他們可能有其他優先順序。ACL 會分離受話方，並轉譯呼叫以符合新服務的語意，因此不需要呼叫者在目前的系統中進行變更。

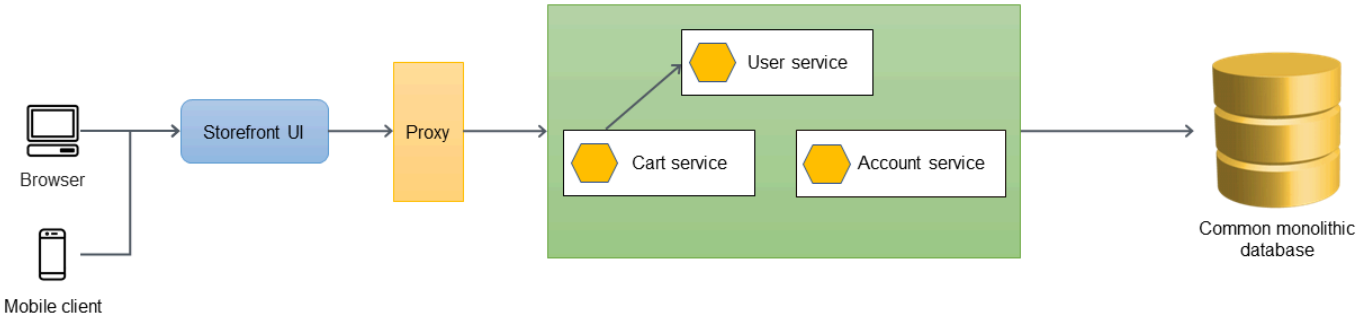
- 操作開銷：ACL 模式需要額外的精力來操作和維護。此工作包括整合 ACL 與監控和警示工具、發行程序，以及持續整合和持續交付 (CI/CD) 程序。
- 單點故障：ACL 中的任何故障都可能導致目標服務無法連線，造成應用程式問題。若要緩解此問題，您應該建置重試功能和斷路器。請參閱使用退避和斷路器模式重試，以進一步了解這些選項。設定適當的提醒和記錄將改善平均解決時間 (MTTR)。
- 技術負債：作為遷移或現代化策略的一部分，請考慮 ACL 是暫時性還是臨時解決方案，還是長期解決方案。如果是臨時解決方案，您應該將 ACL 記錄為技術債務，並在遷移所有相依發起人之後解除委任。
- 延遲：由於將請求從一個界面轉換到另一個界面，額外的 layer 可能會引入延遲。建議您在將 ACL 部署到生產環境之前，在對回應時間敏感的應用程式中定義和測試效能容錯能力。
- 擴展瓶頸：在服務可以擴展到尖峰負載的高負載應用程式中，ACL 可能會成為瓶頸，並可能導致擴展問題。如果目標服務隨需擴展，您應該設計 ACL 來相應擴展。
- 服務特定或共用實作：您可以將 ACL 設計為共用物件，以將呼叫轉換和重新導向至多個服務或服務特定類別。當您判斷 ACL 的實作類型時，請將延遲、擴展和容錯能力納入考量。

實作

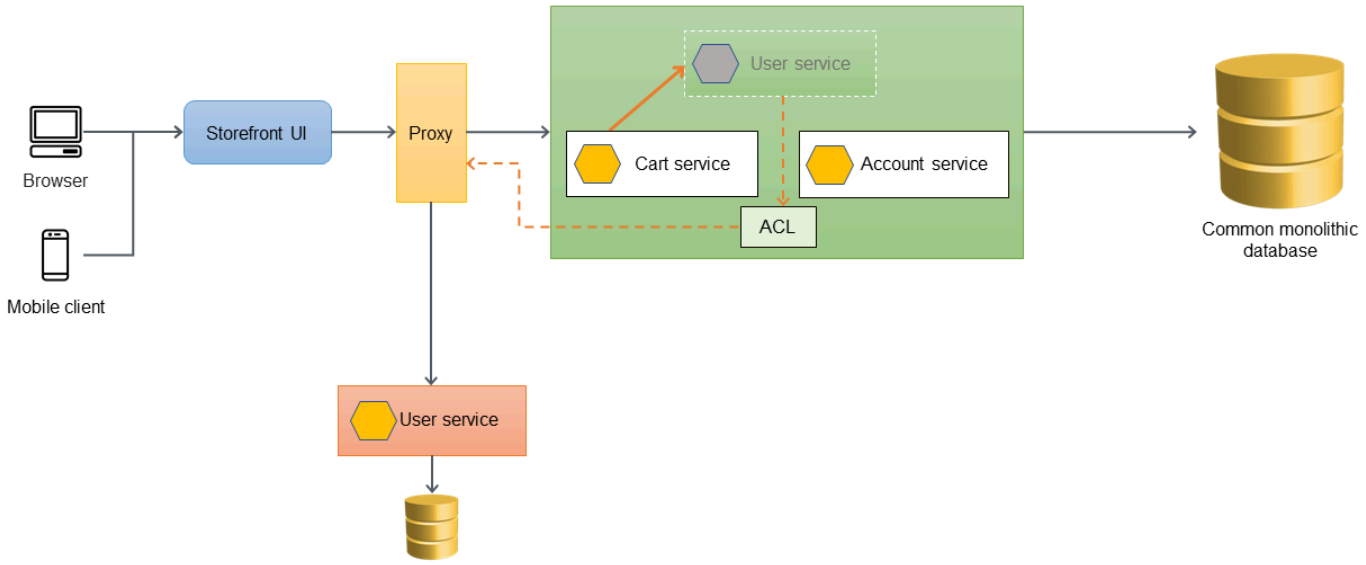
您可以在單體應用程式中實作 ACL，做為要遷移之服務的特定類別，或做為獨立服務。在將所有相依服務遷移至微服務架構之後，必須停用 ACL。

高層級架構

在下列範例架構中，單一應用程式有三種服務：使用者服務、購物車服務和帳戶服務。購物車服務取決於使用者服務，而應用程式會使用單體關聯式資料庫。

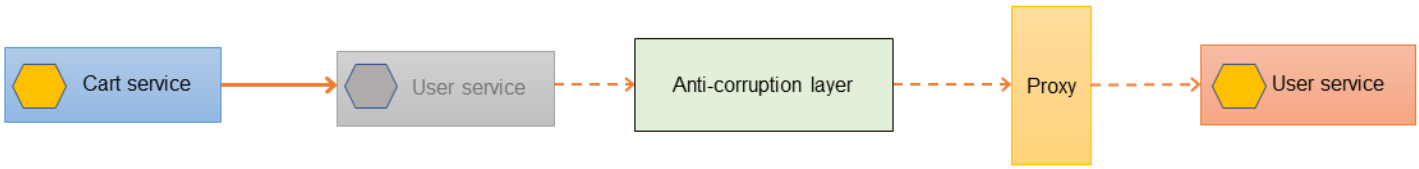


在下列架構中，使用者服務已遷移至新的微服務。購物車服務會呼叫使用者服務，但整體內不再提供實作。 當新遷移服務的界面位於單體應用程式內時，它也可能與其先前的界面不相符。



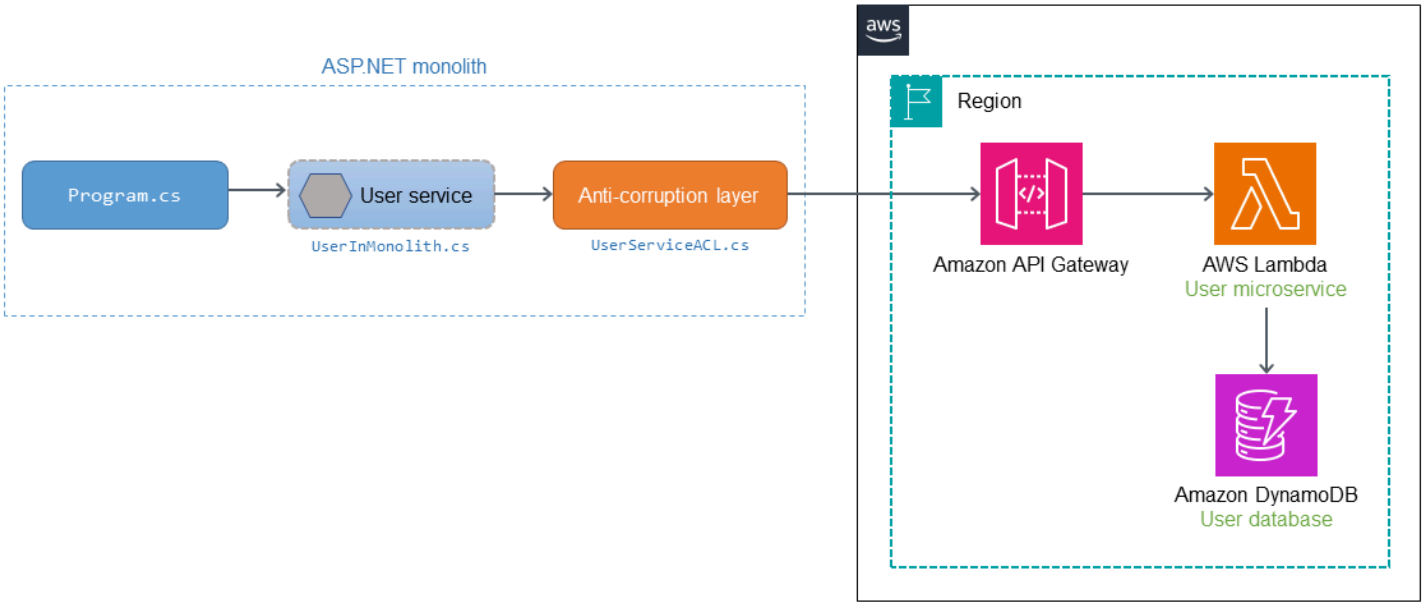
如果購物車服務必須直接呼叫新遷移的使用者服務，這將需要變更購物車服務，並徹底測試整體應用程式。這可能會增加轉型風險和業務中斷。目標是將整體應用程式的現有功能變更降至最低。

在這種情況下，我們建議您在舊使用者服務和新遷移的使用者服務之間引入 ACL。ACL 可做為轉接器或外觀，將呼叫轉換為較新的界面。ACL 可在單體應用程式內實作為 類別 （例如，`UserServiceFacade`或 `UserServiceAdapter`），其專屬於已遷移的服務。所有相依服務遷移至微服務架構後，必須停用反貪汙層。



使用 AWS 服務實作

下圖顯示如何使用 服務實作此 ACL 範例 AWS 。



使用者微服務會從 ASP.NET 整體應用程式遷移，並在 AWS 上部署為 AWS Lambda函數。Lambda 函數的呼叫會透過 Amazon API Gateway 路由。ACL 部署在整體中，以轉譯呼叫，以適應使用者微服務的語意。

當 在整體內部Program.cs呼叫使用者服務 (UserInMonolith.cs) 時，呼叫會路由到 ACL (UserServiceACL.cs)。ACL 會將呼叫轉譯為新的語意和界面，並透過 API Gateway 端點呼叫微服務。發起人 (Program.cs) 不知道使用者服務和 ACL 中發生的轉譯和路由。由於發起人不知道程式碼變更，因此業務中斷和轉型風險較低。

範本程式碼

下列程式碼片段提供原始服務和 實作的變更UserServiceACL.cs。收到請求時，原始使用者服務會呼叫 ACL。ACL 會轉換來源物件以符合新遷移服務的界面、呼叫服務，並將回應傳回給發起人。

```
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 儲存庫

如需此模式的範例架構的完整實作，請參閱 GitHub 儲存庫，網址為 https：//https://github.com/aws-samples/anti-corruption-layer-pattern。

相關內容

- Strangler 無花果模式
- 斷路器模式
- 使用輪詢重試模試

API 路由模式

在敏捷開發環境中，自治團隊 (例如小隊和部落) 擁有一個或多個包含許多微服務的服務。團隊將這些服務公開為 API，以允許其消費者與他們的服務和動作群組進行互動。

使用主機名稱和路徑將 HTTP API 公開給上游消費者有三種主要方法：

方法	描述	範例
主機名稱路由	將每個服務公開為主機名稱。	`billing.api.example.com`
路徑路由	將每個服務公開為路徑。	`api.example.com/billing`
標頭型路由	將每個服務公開為 HTTP 標頭。	`x-example-action: something`

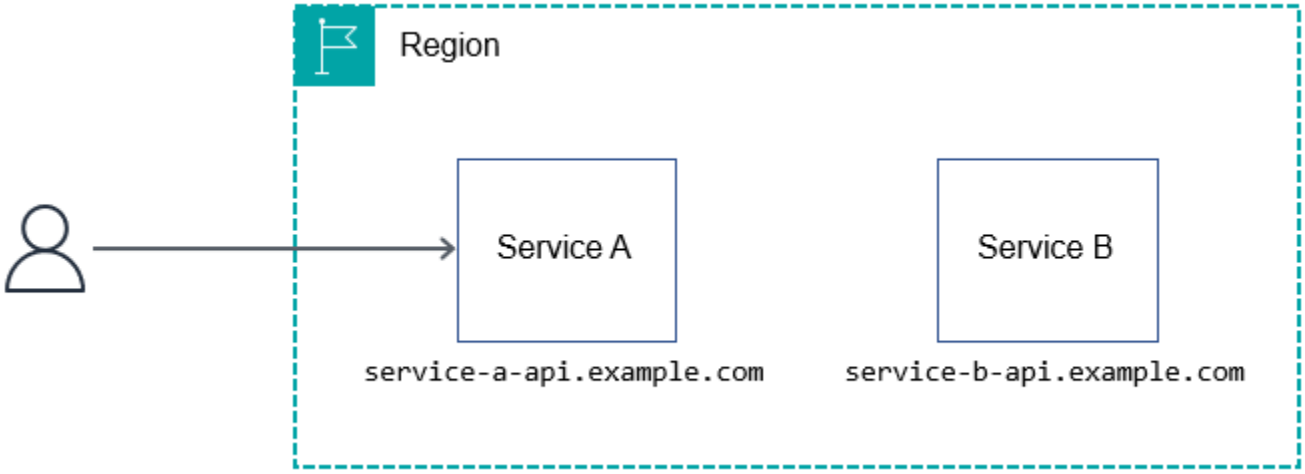
此區段概述了這三種路由方法的典型使用案例及其權衡，以幫助您決定哪種方法最適合您的需求和組織結構。

主機名路由模式

透過主機名稱路由是一種透過為每個 API 提供自己的主機名來隔離 API 服務的機制；例如，`service-a.api.example.com` 或 `service-a.example.com`。

一般使用案例

使用主機名稱進行路由可減少版本中的摩擦量，因為服務團隊之間不會共用任何內容。團隊負責管理從 DNS 項目到生產中的服務操作的所有內容。



優點

主機名稱路由是目前為止最直接且可擴展的 HTTP API 路由方法。您可以使用任何相關的 AWS 服務來建置遵循此方法的架構 – 您可以使用 Amazon API Gateway、AWS AppSync、Application Load Balancer 和 Amazon Elastic Compute Cloud (Amazon EC2) 或任何其他 HTTP 相容服務建立架構。

團隊可以使用主機名稱路由來完全擁有自己的子網域。它也可讓您更輕鬆地隔離、測試和協調特定 AWS 區域 或 版本的部署；例如 `region.service-a.api.example.com`或 `dev.region.service-a.api.example.com`。

缺點

當您使用主機名稱路由時，您的消費者必須記住不同的主機名稱，才能與您公開的每個 API 進行互動。您可以透過提供客戶端 SDK 來緩解此問題。不過，使用者端 SDK 會面臨各自的挑戰。例如，他們必須支援滾動更新，多種語言，版本控制，溝通由安全問題或錯誤修復，文檔等引起的突破性更改。

當您使用主機名稱路由時，也需要在每次建立新服務時註冊子網域或網域。

路徑路由模式

依路徑路由是將多個或所有 API 分組在相同主機名稱下的機制，並使用請求 URI 隔離服務；例如，`api.example.com/service-a` 或 `api.example.com/service-b`。

一般使用案例

大多數團隊選擇這種方法是因為他們想要一個簡單的架構：開發人員只需記住一個 URL，例如與 HTTP API 互動的 `api.example.com`。API 文件通常更容易理解，因為它通常會將資訊集中在一處，而不是分散在不同的入口網站或 PDF。

路徑型路由公認為是共享 HTTP API 的簡單機制。但是，由於多個躍點，它涉及操作開銷，例如組態、授權、整合與其他延遲。它也需要成熟的變更管理程序，以確保設定錯誤不會中斷所有服務。

在 上 AWS，有多種方式可以共用 API 並有效地路由到正確的服務。下列區段將探討三種方法：HTTP 服務反向代理程式、API Gateway 和 Amazon CloudFront。統一 API 服務的建議方法都不依賴於 AWS 執行的下游服務。這些服務可以在任何地方執行沒有問題或任何技術，只要它們是 HTTP 相容。

HTTP 服務反向代理程式

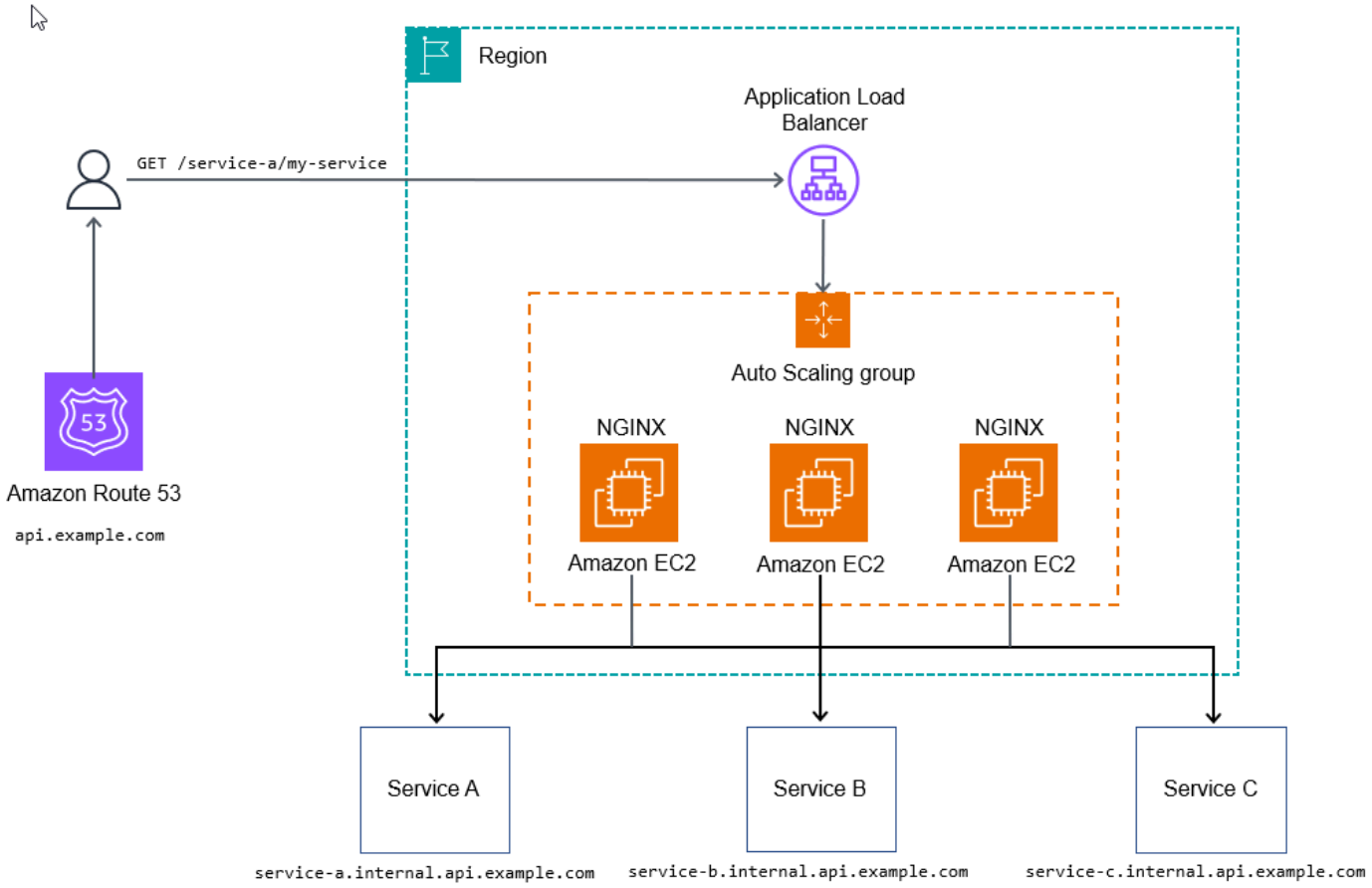
您可以使用 HTTP 伺服器 (例如 NGINX) 來建立動態路由設定。在 Kubernetes 架構中，您也可以建立傳入規則來比對服務的路徑。(本指南不涵蓋 Kubernetes 傳入；請參閱 Kubernetes 文件以瞭解更多資訊。)

NGINX 的下列組態會動態地將 `api.example.com/my-service/` 的 HTTP 請求對應到 `my-service.internal.api.example.com`。

```
server {
    listen  80;

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

下列圖表說明了 HTTP 服務反向代理程式方法。



對於某些不使用其他配置來開始處理請求的使用案例，這種方法可能已足夠，使下游 API 可以收集指標和日誌。

若要為作業生產準備就緒做好準備，您將希望能夠為堆疊的每個層級加入可觀測性、加入其他組態，或加入指令碼來自訂 API 傳入點，藉此允許更進階的功能，例如速率限制或用量 Token。

優點

HTTP 服務反向代理程式方法的最終目的是建立可擴充且可管理的方法，將 API 統一成單一網域，使其對任何 API 消費者看起來都是一致的。此方法也可讓您的服務團隊部署和管理自己的 APIs，在 deployment. AWS managed 服務之後以最低的開銷進行追蹤，例如 AWS X-Ray或 AWS WAF，仍然適用於此處。

缺點

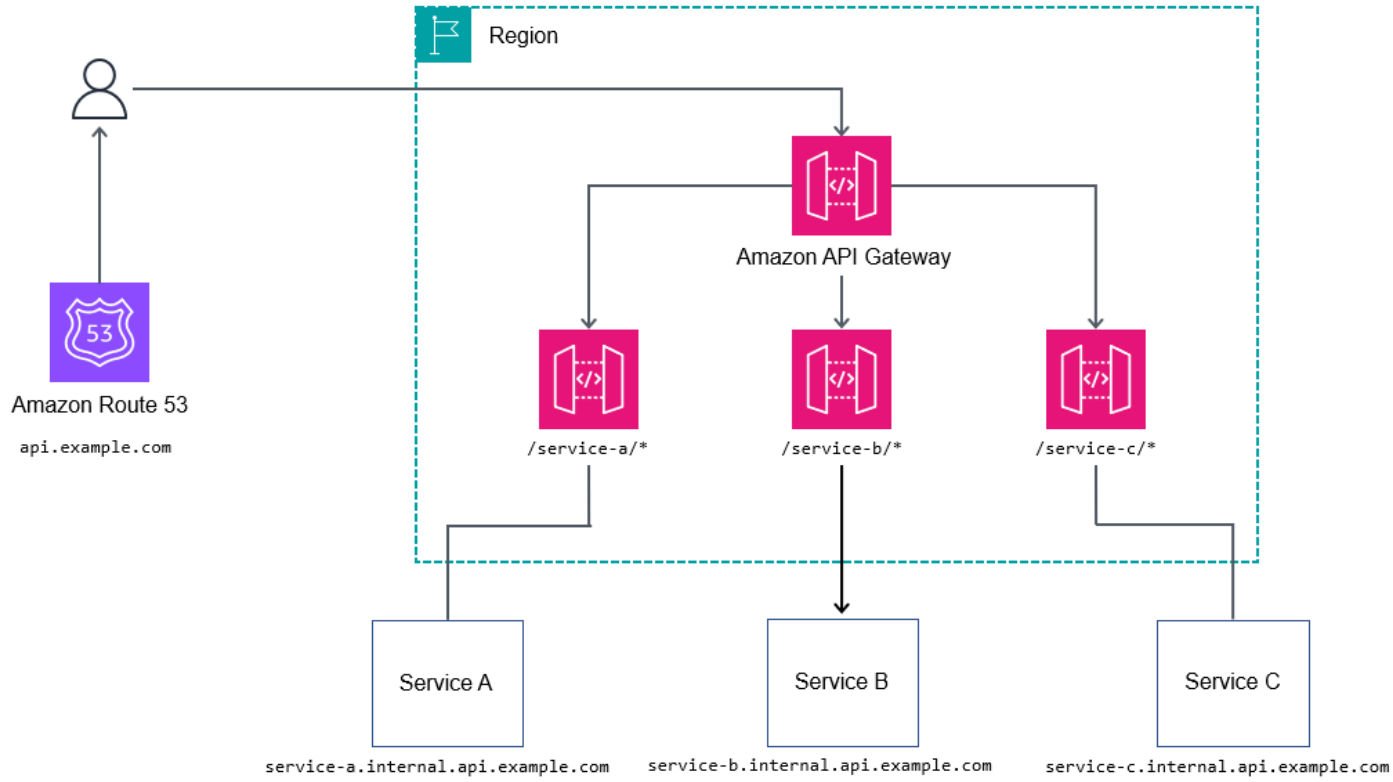
這種方法的主要缺點是對所需的基礎結構元件進行廣泛的測試和管理，不過如果您有網站可靠性工程 (SRE) 團隊，這可能不會是個問題。

這種方法有一個成本臨界點。在低到中等量的情況下，它比本指南中討論的其他一些方法更昂貴。在量大的情況下，它非常具有成本效益 (每秒約 100K 筆交易或更高)。

API Gateway

Amazon API Gateway 服務 (REST API 和 HTTP API) 可以透過類似於 HTTP 服務反向代理程式方法來路由流量。在 HTTP Proxy 模式下使用 API 閘道可提供一種簡單的方法，將許多服務包裝到頂層子網域 `api.example.com` 的進入點中，然後將要求代理至巢狀服務；例如，`billing.internal.api.example.com`。

您可能不希望透過對應根或核心 API 閘道中每個服務中的每個路徑來變得太細微。請改為選擇萬用字元路徑，例如 `/billing/*` 將要求轉寄至帳單服務。如果不對應根或核心 API 閘道中的每個路徑，您就可以在 API 上獲得更大的彈性，因為您不必在每次 API 變更時更新根 API 閘道。



優點

為了控制更複雜的工作流程 (例如變更要求屬性)，REST API 會公開 Apache Velocity 範本語言 (VTL)，讓您修改要求和回應。REST API 可以提供其他好處，例如：

- 使用 AWS Identity and Access Management (IAM)、Amazon Cognito 或授權方驗證 N/Z Amazon Cognito AWS Lambda

- AWS X-Ray 用於追蹤
- 與 整合 AWS WAF
- 基本速率限制
- 將消費者分組到不同層級的用量 Token (請參閱 API Gateway 文件中的限流 API 請求以取得更好的輸送量)

缺點

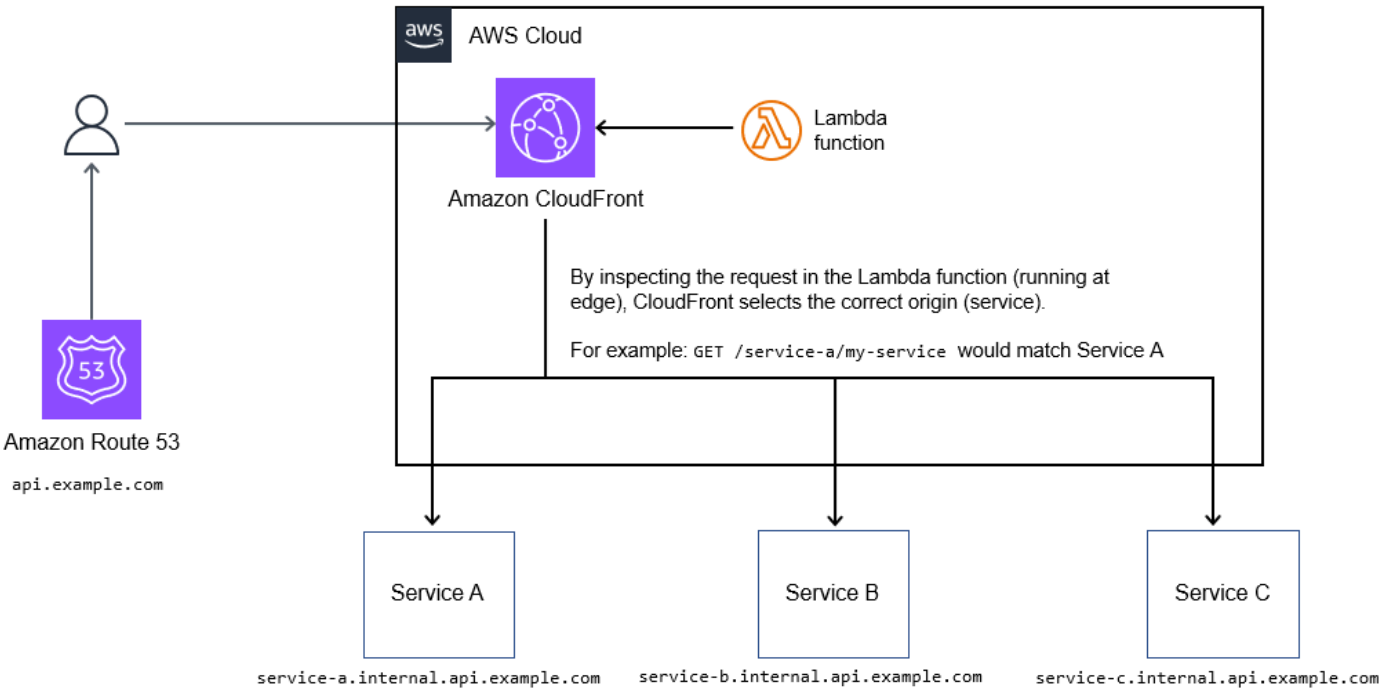
在量大的情況下，成本對部分使用者而言可能是個問題。

CloudFront

您可以使用 Amazon CloudFront 中的動態來源選取功能，有條件地選取要轉寄請求的來源 (服務)。您可以使用此功能，透過單一主機名稱來路由多個服務，例如 `api.example.com`。

一般使用案例

路由邏輯以程式碼形式存放於 Lambda @Edge 函數中，因此它支援高度可自訂的路由機制，例如 A/B 測試、初期測試版本、功能標記和路徑重新寫入。這會在下列圖表中進行說明。



優點

如果您需要快取 API 回應，此方法是在單一端點後方統一服務集合的好方法。這是統一 API 集合的具有成本效益的方法。

此外，CloudFront 支援欄位層級加密，以及與 整合 AWS WAF ，以實現基本速率限制和基本 ACLs。

缺點

此方法最多支援 250 個可以統一的原始伺服器 (服務)。對於大多數部署而言，此限制已足夠，但隨著服務組合的增加，可能會導致大量 API 發生問題。

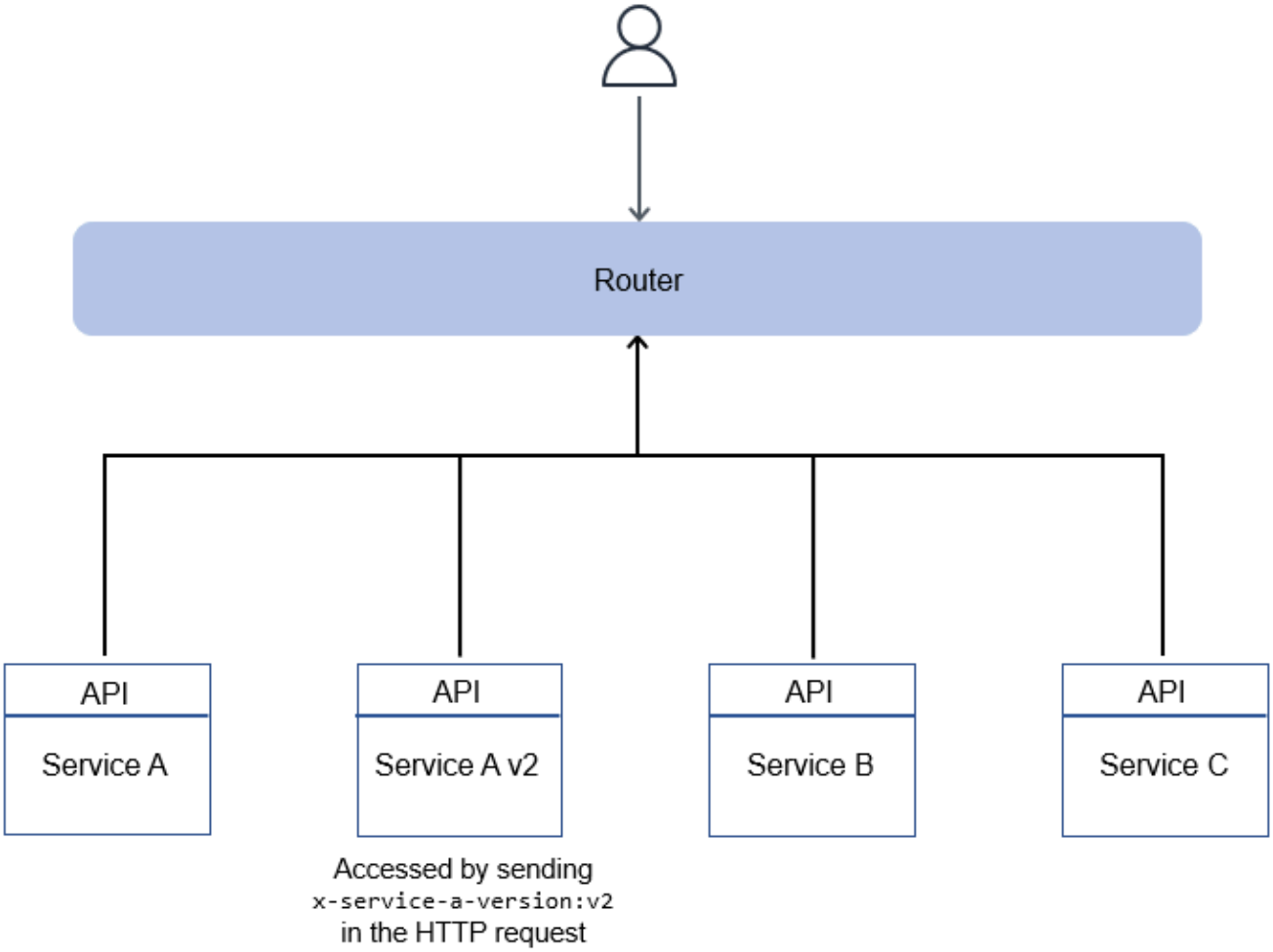
目前更新 Lambda @Edge 函數需要幾分鐘的時間。CloudFront 還需要長達 30 分鐘的時間來完成將變更傳播到所有存在點。這最終會阻止進一步更新，直到它們完成。

HTTP 標頭路由模式

標頭型路由可讓您透過在 HTTP 請求中指定 HTTP 標頭來鎖定每個請求的正確服務。例如，傳送標頭 `x-service-a-action: get-thing` 能讓您從 `Service A` 來 `get thing`。請求的路徑仍然很重要，因為它提供了有關您嘗試使用哪種資源的指導。

除了對動作使用 HTTP 標頭路由之外，您還可以將其用作版本路由的機制，啟用功能旗標、A/B 測試或類似需求。實際上，您可能會使用標頭路由與其他路由方法其中之一來建立強大的 API。

HTTP 標頭路由的架構通常在微服務前有一個精簡路由層，該層會路由到正確的服務並回傳回應，如下列圖表所示。此路由層可以涵蓋所有服務或只是一些服務，以啟用作業，例如版本型路由。



優點

組態變更只需要很少的精力，而且可以輕鬆自動化。此方法也很靈活，並且支援創造性的方式，以僅公開您希望從服務中獲得的特定操作。

缺點

與主機名稱路由方法一樣，HTTP 標頭路由會假設您對用戶端擁有完全控制權，而且可以操作自訂 HTTP 標頭。代理伺服器、內容交付網路 (CDN) 和負載平衡器可以限制標頭大小。雖然不太需要擔憂，但根據您增加的標頭和 Cookie 數量，這有可能變成問題。

斷路器模式

意圖

當呼叫先前造成重複逾時或失敗時，斷路器模式可防止發起人服務重試對其他服務 (來電者) 的呼叫。模式也會用來偵測受話方服務何時再次運作。

動機

當多個微服務協同處理請求時，一或多個服務可能會變得無法使用或呈現高延遲。當複雜的應用程式使用微服務時，一個微服務中的中斷可能會導致應用程式失敗。微服務會透過遠端程序呼叫進行通訊，而網路連線中可能發生暫時性錯誤，進而導致失敗。（暫時性錯誤可以透過使用具有退避模式的重試來處理。）在同步執行期間，逾時或失敗的層疊可能會導致使用者體驗不佳。

不過，在某些情況下，故障可能需要更長的時間才能解決，例如，當受話方服務或資料庫爭用導致逾時時。在這種情況下，如果呼叫服務重複重試呼叫，這些重試可能會導致網路爭用和資料庫執行緒集區耗用。此外，如果多個使用者重複重試應用程式，這會使問題惡化，並可能導致整個應用程式的效能降低。

Michael Nygard 在他的書籍 Release It (Nygard 2018) 中熱門斷路器模式。此設計模式可防止發起人服務重試先前導致重複逾時或失敗的服務呼叫。它也可以偵測受話方服務何時再次運作。

斷路器物件的運作方式類似斷路器，會在電路異常時自動中斷目前的 。發生故障時，斷路器會關閉或觸發目前的流程。同樣地，斷路器物件位於發起人和受話方服務之間，如果受話方無法使用，則會跳轉。

分散式運算的缺點是 Peter Deutsch 和其他在 Sun Microsystems 所做的一組聲明。他們說初次接觸分散式應用程式的程式設計人員，一定會做出錯誤的假設。網路可靠性、零延遲預期和頻寬限制會導致軟體應用程式以最少的錯誤處理方式寫入網路錯誤。

在網路中斷期間，應用程式可能會無限期等待回覆並持續使用應用程式資源。在網路可用時未重試操作也可能導致應用程式降級。如果 API 因為網路問題而呼叫資料庫或外部服務逾時，沒有斷路器的重複呼叫可能會影響成本和效能。

適用性

在下列情況下使用此模式：

- 呼叫者服務進行最有可能失敗的呼叫。
- 受話方服務呈現的高延遲（例如，當資料庫連線緩慢時）會導致呼叫者服務逾時。
- 呼叫者服務會進行同步呼叫，但呼叫者服務無法使用或顯示高延遲。

問題和考量

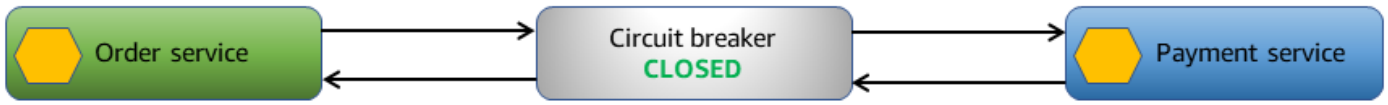
- 服務無關實作：為了防止程式碼膨脹，建議您以微服務無關和 API 驅動的方式實作斷路器物件。
- 受話方電路關閉：當受話方從效能問題或失敗中復原時，他們可以將電路狀態更新為 CLOSED。這是斷路器模式的延伸，如果您的復原時間目標 (RTO) 需要，則可以實作。
- 多執行緒呼叫：過期逾時值定義為在再次路由呼叫之前，電路保持跳閘的期間，以檢查服務可用性。在多個執行緒中呼叫受話方服務時，失敗的第一個呼叫會定義過期逾時值。您的實作應確保後續呼叫不會無限移動過期逾時。
- 強制開啟或關閉電路：系統管理員應該能夠開啟或關閉電路。這可以透過更新資料庫資料表中的過期逾時值來完成。
- 可觀測性：應用程式應設定記錄，以識別斷路器開啟時失敗的呼叫。

實作

高層級架構

在下列範例中，發起人是訂單服務，來電者是付款服務。

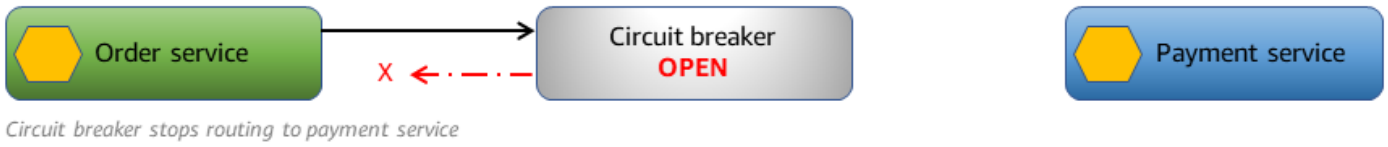
當沒有失敗時，訂單服務會將斷路器的所有呼叫路由至付款服務，如下圖所示。



如果付款服務逾時，斷路器可以偵測逾時並追蹤失敗。

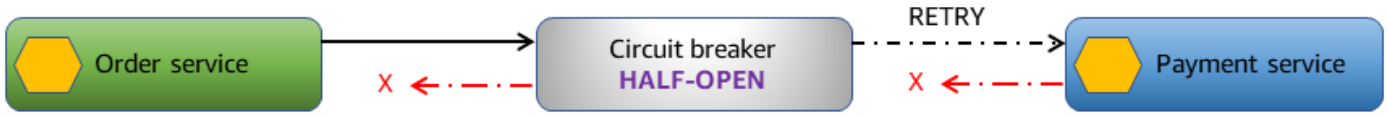
Circuit breaker with payment service failure

如果逾時超過指定的閾值，應用程式會開啟電路。當電路開啟時，斷路器物件不會將呼叫路由到付款服務。當訂單服務呼叫付款服務時，它會傳回立即失敗。



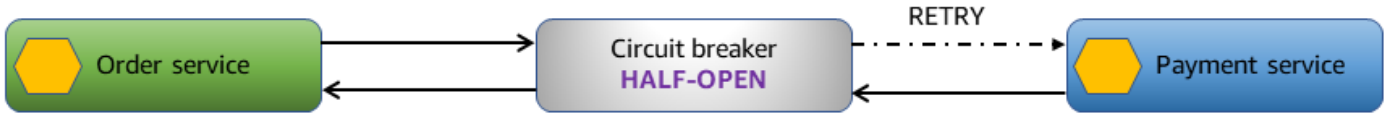
Circuit breaker stops routing to payment service

斷路器物件會定期嘗試查看對付款服務的呼叫是否成功。



Circuit breaker periodically retries payment service

當呼叫付款服務成功時，電路會關閉，所有進一步的呼叫都會再次路由到付款服務。



Circuit breaker with working payment service

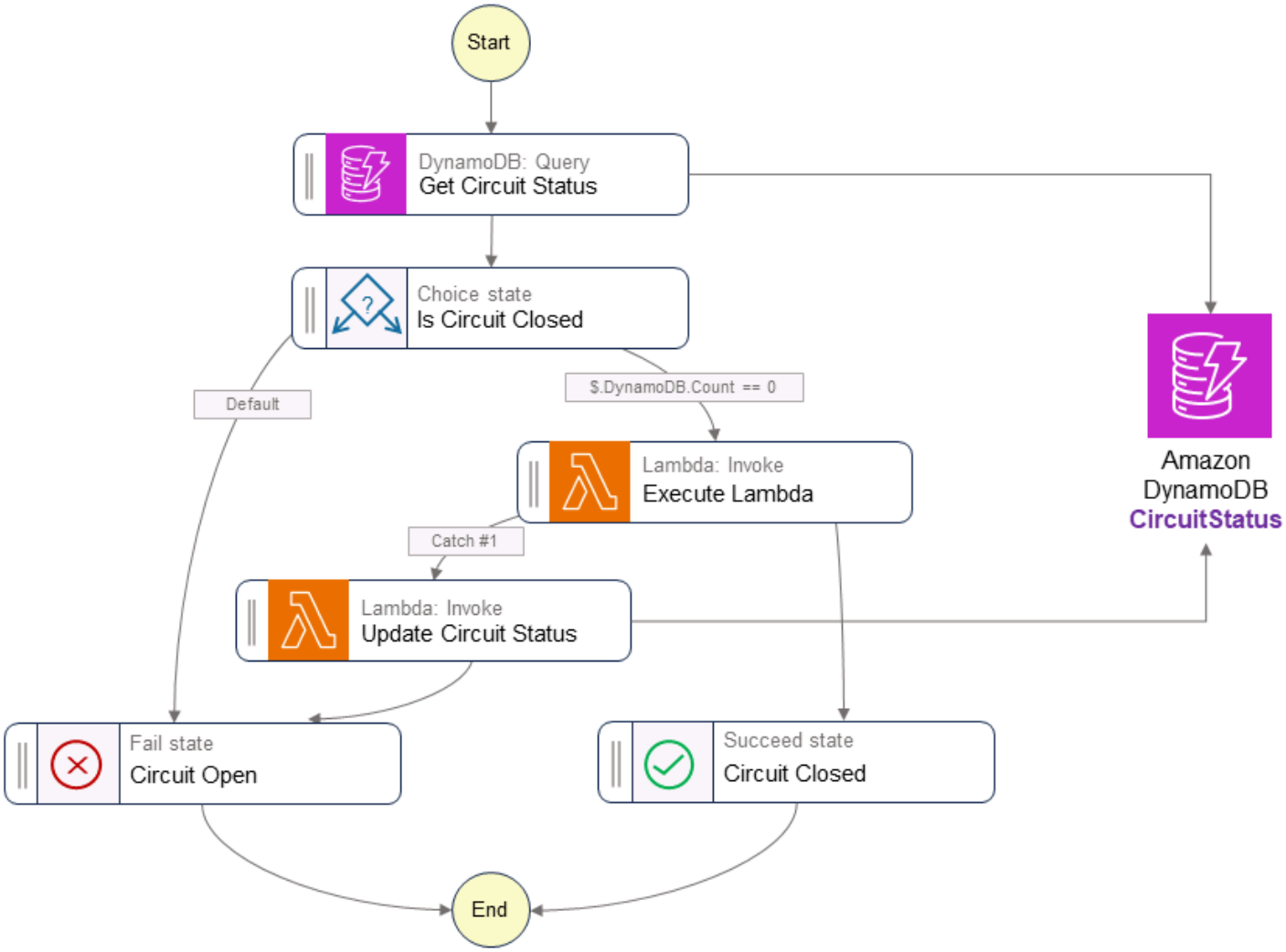
使用 AWS 服務實作

範例解決方案使用 中的快速工作流程AWS Step Functions來實作斷路器模式。Step Functions 狀態機器可讓您設定模式實作所需的重試功能和決策型控制流程。

解決方案也會使用 Amazon DynamoDB 資料表做為資料存放區來追蹤電路狀態。這可以替換為記憶體內資料存放區，例如 Amazon ElastiCache (Redis OSS)，以獲得更好的效能。

當服務想要呼叫另一個服務時，它會以受話方服務的名稱啟動工作流程。工作流程會從 DynamoDB `CircuitStatus`資料表取得斷路器狀態，該資料表存放目前降級的服務。如果 `CircuitStatus`包含受話方的未過期記錄，表示電路已開啟。Step Functions 工作流程會傳回立即失敗並結束 `FAIL` 狀態。

如果`CircuitStatus`資料表不包含受話方的記錄或包含過期的記錄，則服務會正常運作。狀態機器定義的`ExecuteLambda`步驟會呼叫透過參數值傳送的 Lambda 函數。如果呼叫成功，Step Functions 工作流程會以 `SUCCESS` 狀態結束。



如果服務呼叫失敗或發生逾時，應用程式會在定義的次數內以指數退避重試。如果服務呼叫在重試後失敗，工作流程會將記錄插入具有 的 服務`CircuitStatus`資料表中`ExpiryTimeStamp`，而工作流程會以 FAIL 狀態結束。只要斷路器開啟，對相同服務的後續呼叫就會立即傳回故障。狀態機器定義的`Get Circuit Status`步驟會根據 `ExpiryTimeStamp`值檢查服務可用性。過期的項目會使用 DynamoDB 存留時間 (TTL) 功能從`CircuitStatus`資料表中刪除。

範本程式碼

下列程式碼使用 `GetCircuitStatus` Lambda 函數來檢查斷路器狀態。

```
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 = "";
}
```

下列程式碼顯示 Step Functions 工作流程中的 Amazon States Language 陳述式。

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

GitHub 儲存庫

如需此模式範例架構的完整實作，請參閱 GitHub 儲存庫，網址為 https：//https://github.com/aws-samples/circuit-breaker-netcore-blog。

部落格參考

- 搭配 AWS Step Functions 和 Amazon DynamoDB 使用斷路器模式

相關內容

- Strangler 無花果模式
- 使用輪詢重試模試

事件來源模式

意圖

在事件驅動架構中，事件來源模式會將導致狀態變更的事件儲存在資料存放區中。這有助於捕獲和維護狀態變化的完整歷史記錄，並提高可稽核性、可追溯性和分析過去狀態的能力。

動機

多個微服務可以協同合作以處理請求，並透過事件進行通訊。這些事件可能會導致狀態 (資料) 發生變更。依照事件物件發生的順序儲存，可提供有關資料實體目前狀態的重要資訊，以及其到達該狀態的其他相關資訊。

適用性

在下列情況下使用事件來源模式：

- 追蹤需要應用程式中發生之事件的不可變歷程記錄。
- 必須提供來自單一事實來源 (SSOT) 的多語言資料預測。
- 需要對應用程式狀態進行指向時間重建。
- 不需要長期存儲應用程式狀態，但您可能需要根據需要重建它。
- 工作負載具有不同的讀寫磁碟區。例如，您的寫入密集型工作負載不需要即時處理。
- 必須具備變更資料擷取 (CDC) 才能分析應用程式效能和其他指標。
- 系統中發生的所有事件都需要稽核資料，以便進行報告和法規遵循。
- 您想要在重新執行程序期間變更 (插入、更新或刪除) 事件來衍生假設案例，以判斷可能的結束狀態。

問題和考量

- 樂觀並行控制：此模式存儲導致系統中狀態更改的每個事件。多個使用者或服務可以嘗試同時更新相同的資料片段，造成事件衝突。這些衝突發生在同一時間建立並套用衝突的事件時，這會導致與實際不符的最終資料狀態。若要解決此問題，您可以實作偵測和解決事件衝突的策略。例如，您可以透過包含版本控制或將時間戳記新增至事件以追蹤更新順序，以實作樂觀的並行控制結構描述。

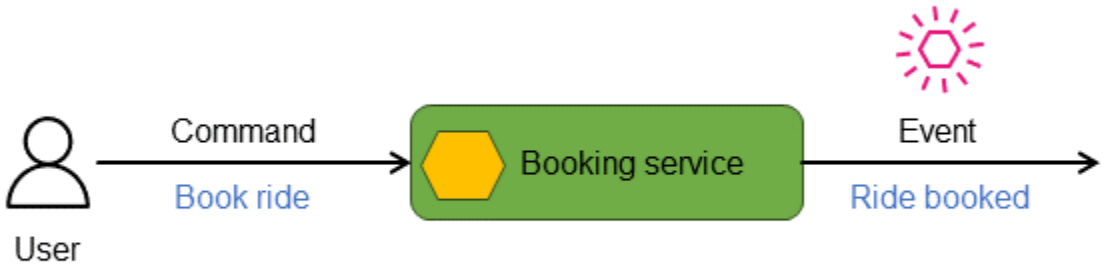
- 複雜性：實施事件採購需要將心態從傳統的 CRUD 操作轉變為事件驅動的思維。用來將系統還原至原始狀態的重新執行程序可能很複雜，以確保資料等冪性。事件儲存、備份和快照也會增加額外的複雜性。
- 最終一致性：由於使用命令查詢職責隔離 (CQRS) 模式或具體化視觀表更新資料時，資料預測最終會保持一致。當消費者處理來自事件存放區的資料，而發布者傳送新資料時，資料投影或應用程式物件可能不代表目前的狀態。
- 查詢：與傳統資料庫相比，從事件日誌擷取目前或彙總資料可能會更複雜且速度較慢，尤其是複雜的查詢和報告工作。為了緩解此問題，事件來源通常會使用 CQRS 模式來實作。
- 事件存放區的大小和成本：隨著事件持續持續存在，事件存放區可能會經歷指數級的大小成長，尤其是在具有高事件輸送量或延長保留期的系統中。因此，您必須定期將事件資料封存至符合成本效益的儲存體，以防止事件存放區變得過大。
- 事件存放區的可擴展性：事件存放區必須有效率地處理大量的寫入和讀取作業。擴展事件存放區可能具有挑戰性，因此擁有提供碎片和分區的資料存放區非常重要。
- 效率和最佳化：選擇或設計可有效處理寫入和讀取作業的事件存放區。事件存放區應針對應用程式的預期事件磁碟區和查詢模式進行最佳化。實施索引和查詢機制可以在重建應用程式狀態時加快事件的擷取速度。您也可以考慮使用專門的事件存放區資料庫或提供查詢最佳化功能的程式庫。
- 快照：您必須以時間為基礎的啟用定期備份事件日誌。在上次成功備份資料時重新執行事件，應該會導致應用程式狀態的時間點復原。復原點目標 (RPO) 是自上次資料復原點以來可接受的時間上限。RPO 會判斷最後一個復原點與服務中斷之間可接受的資料遺失。資料和事件存放區的每日快照頻率應以應用程式的 RPO 為基礎。
- 時間敏感度：事件會依照事件發生的順序儲存。因此，網路可靠性是實作此模式時要考慮的重要因素。延遲問題可能會導致系統狀態不正確。使用先進先出 (FIFO) 佇列 (最多一次傳遞)，將事件傳送至事件存放區。
- 事件重新顯示效能：重新執行大量事件以重建目前的應用程式狀態可能非常耗時。為了增強效能，尤其是從封存資料重新播放事件時，必須進行最佳化工作。
- 外部系統更新：使用事件來源模式的應用程式可能會更新外部系統中的資料存放區，而且可能會將這些更新擷取為事件物件。在活動重播期間，如果外部系統預期不會更新，這可能會成為問題。在這種情況下，您可以使用功能旗標來控制外部系統更新。
- 外部系統查詢：當外部系統調用對呼叫的日期和時間敏感時，接收的資料可以存儲在內部資料存儲中，以便在重播期間使用。
- 事件版本控制：隨著應用程式的發展，事件的結構 (結構描述) 可能會改變。實作事件的版本控制策略，以確保向後和向前相容性是必要的。這可能涉及在事件裝載中包含版本欄位，以及在重新執行期間適當地處理不同的事件版本。

實作

高層級架構

命令和事件

在分散式、事件驅動的微服務應用程式中，命令代表傳送至服務的指示或要求，通常是在其狀態中啟動變更的目的。服務會處理這些命令，並評估指令對其目前狀態的有效性和適用性。如果命令執行成功，服務會透過發出表示採取的動作和相關狀態資訊的事件進行回應。例如，在下圖中，預訂服務會透過發出已預訂「乘車」事件來回應「預訂」乘車命令。



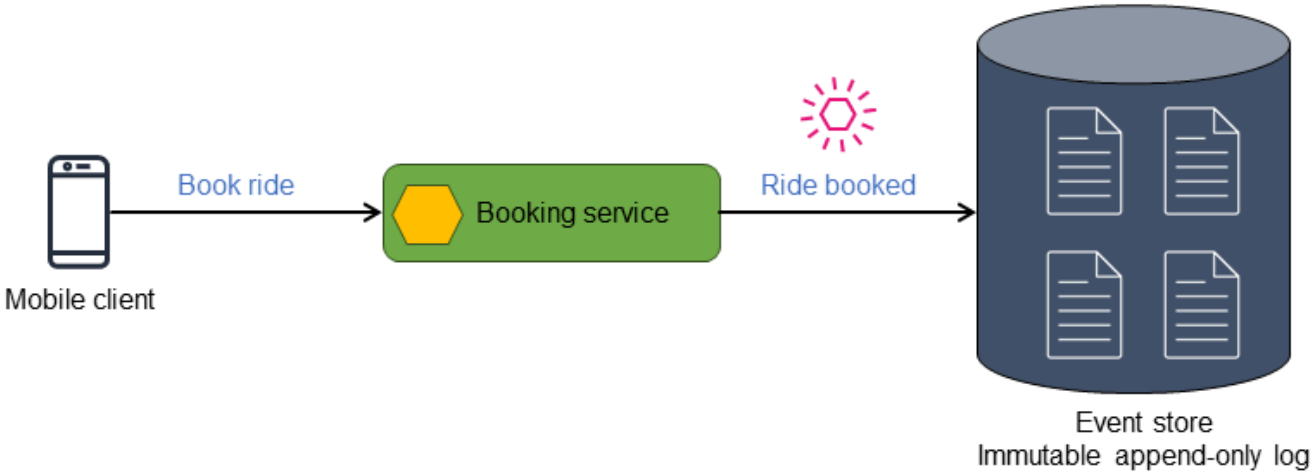
事件存放區

事件會記錄到不可變、僅附加、按時間順序排序的儲存庫或稱為事件存放區的資料存放區中。每個狀態變更都會視為個別事件物件。具有已知初始狀態、其目前狀態以及任何時間點檢視的實體物件或資料存放區，可以按照事件發生的順序重新建構。

事件存儲作為所有操作和狀態變化的歷史記錄，並作為有價值的單一事實來源。您可以透過重新執行處理器傳遞事件，使用事件存放區來衍生系統的最終最新狀態，重新執行處理器會套用這些事件，以產生最新系統狀態的精確表示。您也可以透過重新執行處理器重新播放事件，使用事件存放區來產生狀態的時間點透視。在事件來源模式中，目前狀態可能不會完全由最新的事件物件來表示。有三種方式可以得出：

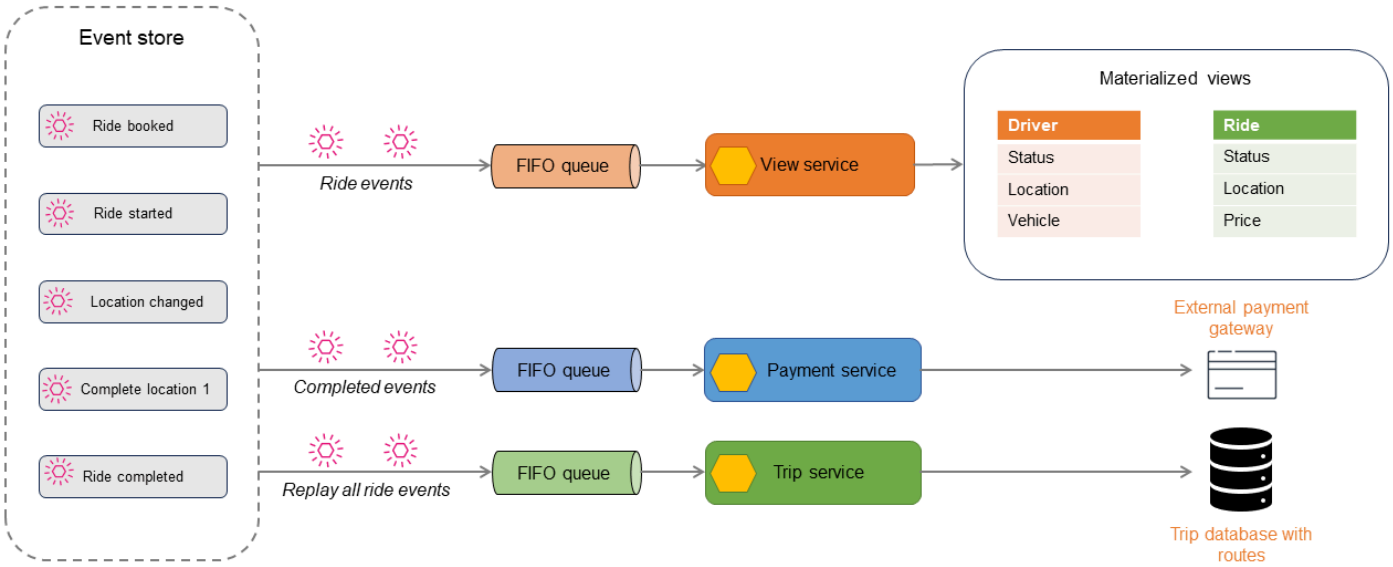
- 透過彙總相關事件的方式。相關的事件物件會結合在一起，以產生用於查詢的目前狀態。這種方法通常與 CQRS 模式結合使用，因為事件會合併並寫入唯讀資料存放區。
- 透過使用具體化視觀表的方式。您可以使用具體化視觀表模式的事件來源來計算或彙總事件資料，並取得相關資料的目前狀態。
- 透過重播事件的方式。事件物件可重新顯示，以執行產生目前狀態的動作。

下圖顯示要儲存在 Ride booked 事件存放區中的事件。



事件存放區會發布其儲存的事件，並可篩選並將事件路由至適當的處理器，以供後續動作使用。例如，可將事件路由至總結狀態並顯示具體化視觀表的視觀表處理器。事件會轉換為目標資料倉儲的資料格式。這種架構可加以延伸以導出不同類型的資料存放區，導致資料的多語言持續性。

下列圖表說明乘車預訂應用程式中的事件。應用程式內發生的所有事件都會儲存在事件存放區中。然後會篩選儲存的事件，並將其路由至不同的消費者。



乘車事件可用於透過使用 CQRS 或具體化視觀表模式產生唯讀資料存放區。您可以透過查詢讀取商店來取得乘車、駕駛或預訂的目前狀態。某些事件 (例如 Location changed 或 Ride completed)

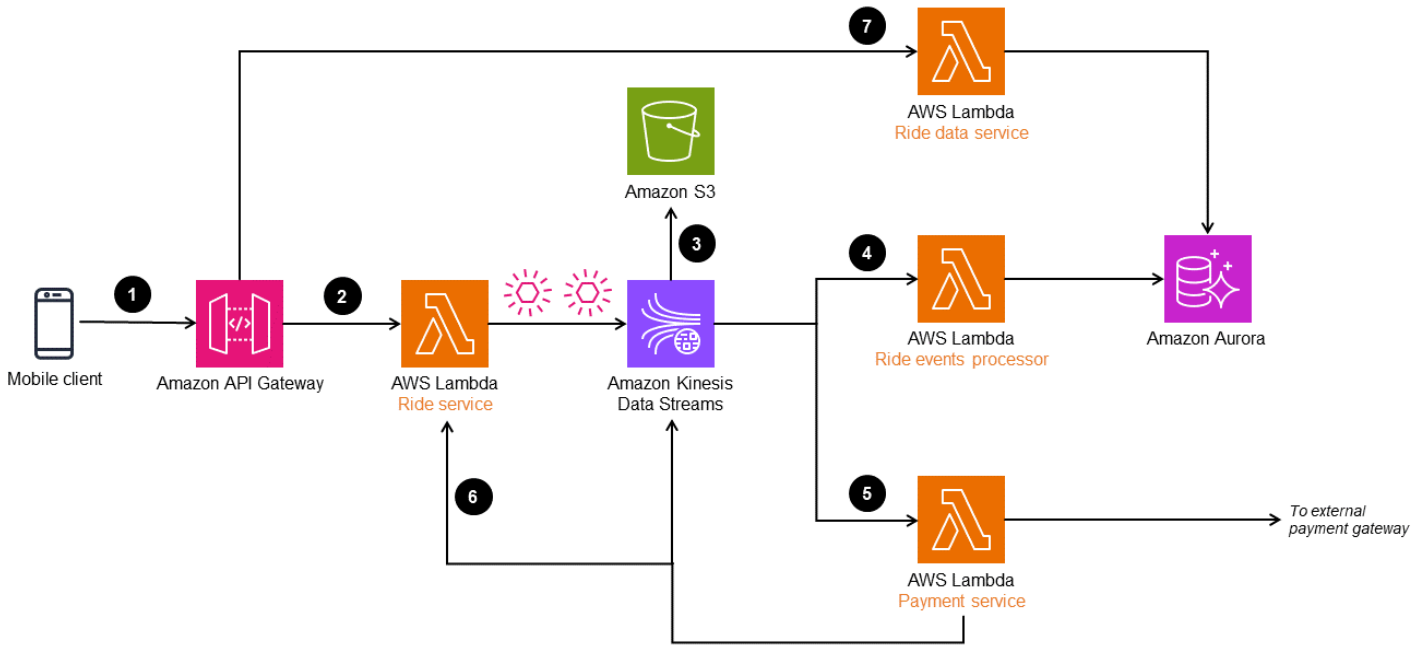
會發布給其他消費者進行付款處理。乘車完成後，系統會重新顯示所有乘車事件以建立乘車歷史記錄，以供稽核或報告之用。

事件來源模式通常用於需要時間點復原的應用程式中，以及必須使用單一事實來源以不同格式預測資料時。這兩項作業都需要重新顯示處理作業，才能執行事件並衍生必要的結束狀態。重新顯示處理器也可能需要一個已知的起點，最好不是從應用程式啟動開始，因為這不是一個有效率的程序。我們建議您定期建立系統狀態的快照，並套用較少數量的事件來衍生最新狀態。

使用 AWS 服務來實作

在下列架構中，Amazon Kinesis Data Streams 會用作事件存放區。此服務將應用程式變更擷取並管理為事件，並提供高輸送量和即時資料串流解決方案。若要在 AWS 上實作事件採購模式，您也可以根據應用程式的需求，使用 Amazon EventBridge 和 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 等服務。

若要增強耐久性並啟用稽核，您可以將 Kinesis Data Streams 擷取的事件封存在 Amazon Simple Storage Service (Amazon S3) 之中。這種雙儲存方法有助於安全地保留歷史事件資料，以供未來分析和法規遵循之用。



工作流程由以下步驟組成：

1. 一個乘車預訂請求透過行動用戶端送出至 Amazon API Gateway 端點。
2. 乘車微服務 (`Ride service` Lambda 函數) 會接收請求、轉換物件，然後發布至 Kinesis Data Streams。

3. Kinesis 資料串流中的事件資料存放在 Amazon S3 中，以達到法規遵循和稽核歷史記錄的目的。
4. 這些事件由 `Ride event processor` Lambda 函數轉換和處理，並存放在 Amazon Aurora 資料庫中，以提供乘車資料的具體化視觀表。
5. 系統會篩選已完成的乘車事件，並將其傳送至外部付款閘道進行付款處理。付款完成後，系統會將另一個事件傳送至 Kinesis Data Streams 以更新乘車資料庫。
6. 乘車完成後，乘車事件會重新提供給 `Ride service` Lambda 函數，以建立路線和乘車歷史記錄。
7. 乘車資訊可以透過 `Ride data service` 讀取，該服務會從 Aurora 資料庫中讀取。

API Gateway 也可以將事件物件直接傳送到 Kinesis Data Streams，而不需使用 `Ride service` Lambda 函數。但是，在複雜的系統中，例如乘車招呼服務，事件物件可能需要先處理和豐富，然後才能擷取到資料串流中。基於這個原因，此架構具有 `Ride service`，可在將事件傳送至 Kinesis Data Streams 之前先處理該事件。

部落格參考

- AWS Lambda 的新功能 — 將 SQS FIFO 做為事件來源

六角形架構模式

意圖

六邊形架構模式也稱為連接埠和轉接器模式，由 Alistair Cockburn 醫生於 2005 年提出。它旨在建立鬆耦合的架構，其中應用程式元件可以獨立測試，而不需要依賴資料存放區或使用者介面 UIs)。此模式有助於防止資料存放區和 UIs 的技術鎖定。這可讓您更輕鬆地隨著時間變更技術堆疊，而對商業邏輯的影響有限或沒有影響。在此鬆耦合架構中，應用程式會透過稱為連接埠的界面與外部元件通訊，並使用轉接器來翻譯與這些元件的技術交換。

動機

六邊形架構模式用於隔離業務邏輯 （網域邏輯） 與相關基礎設施程式碼，例如存取資料庫或外部 APIs 程式碼。此模式適用於為需要與外部服務整合的 AWS Lambda 函數建立鬆散耦合的商業邏輯和基礎設施程式碼。在傳統架構中，常見的做法是將商業邏輯嵌入資料庫層，做為預存程序和使用者介面。此實務以及在商業邏輯中使用 UI 特定的建構，會導致密切結合的架構，在資料庫遷移和使用者體驗 (UX) 現代化工作中造成瓶頸。六邊形架構模式可讓您根據用途而非技術來設計系統和應用程式。此策略可讓您輕鬆交換應用程式元件，例如資料庫、UX 和服務元件。

適用性

在下列情況下使用六邊形架構模式：

- 您想要解耦應用程式架構，以建立可完整測試的元件。
- 多種類型的用戶端可以使用相同的網域邏輯。
- 您的 UI 和資料庫元件需要不會影響應用程式邏輯的定期技術重新整理。
- 您的應用程式需要多個輸入提供者和輸出消費者，而自訂應用程式邏輯會導致程式碼複雜性和缺乏可擴展性。

問題和考量

- 網域驅動型設計： 六角形架構特別適用於網域驅動型設計 (DDD)。每個應用程式元件都代表 DDD 中的子網域，而六邊形架構可用於實現應用程式元件之間的鬆散耦合。
- 可測試性：根據設計，六邊形架構使用抽象處理輸入和輸出。因此，由於固有的鬆耦合，撰寫單元測試和獨立測試變得更容易。

- 複雜性：仔細處理時，將商業邏輯與基礎設施程式碼分開的複雜性，可以帶來敏捷性、測試涵蓋範圍和技術適應性等巨大優勢。否則，問題可能會變得複雜，無法解決。
- 維護開銷：只有在應用程式元件需要多個輸入來源和輸出目的地才能寫入，或輸入和輸出資料存放區必須隨時間變更時，才算有讓架構可插入 的額外轉接器程式碼。否則，轉接器會成為要維護的另一層，這會導致維護開銷。
- 延遲問題：使用連接埠和轉接器會新增另一層，這可能會導致延遲。

實作

六角架構支援隔離應用程式和商業邏輯與基礎設施程式碼，以及整合應用程式與 UIs、外部 APIs、資料庫和訊息中介裝置的程式碼。您可以透過連接埠和轉接器，輕鬆將商業邏輯元件連接到應用程式架構中的其他元件（例如資料庫）。

連接埠是與技術無關的應用程式元件進入點。這些自訂界面會決定允許外部演員與應用程式元件通訊的界面，無論誰或什麼實作了界面。這類似於 USB 連接埠允許許多不同類型的裝置與電腦通訊的方式，只要它們使用 USB 轉接器即可。

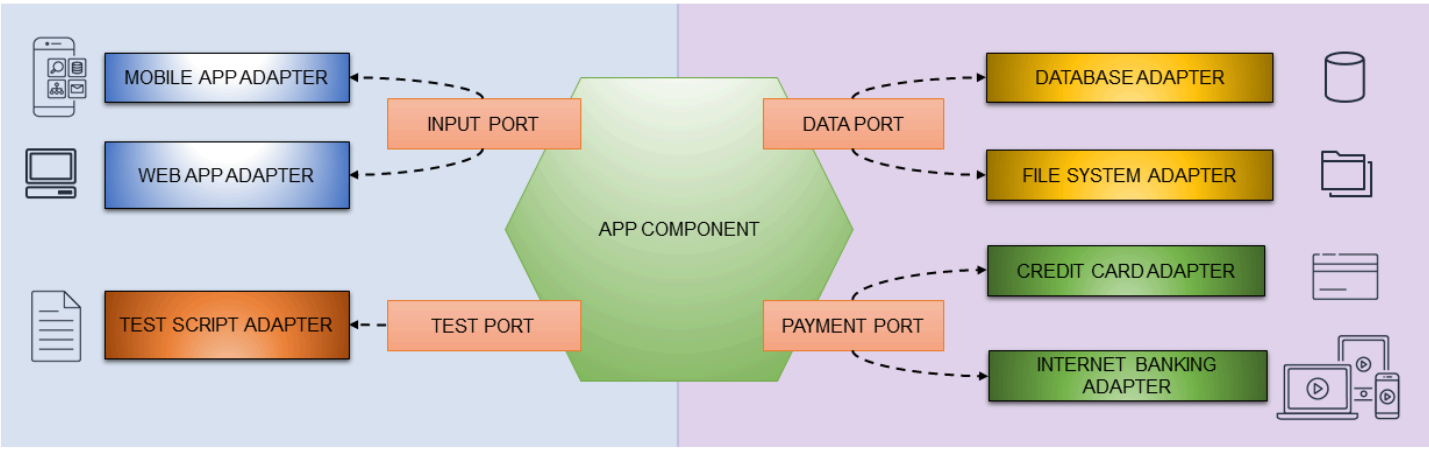
轉接器會使用特定技術，透過連接埠與應用程式互動。轉接器會插入這些連接埠、接收資料或將資料提供給連接埠，以及轉換資料以進行進一步處理。例如，REST 轉接器可讓演員透過 REST API 與應用程式元件通訊。連接埠可以有多個轉接器，而不會對連接埠或應用程式元件造成任何風險。若要延伸上述範例，將 GraphQL 轉接器新增至相同的連接埠，可為演員提供透過 GraphQL API 與應用程式互動的額外方式，而不會影響 REST API、連接埠或應用程式。

連接埠會連線至應用程式，而轉接器可做為外部世界的連線。您可以使用連接埠來建立鬆耦合的應用程式元件，並透過變更轉接器來交換相依元件。這可讓應用程式元件與外部輸入和輸出互動，而不需要具備任何內容感知。元件可在任何層級交換，這有助於自動化測試。您可以獨立測試元件，而不需依賴基礎設施程式碼，而不需佈建整個環境來執行測試。應用程式邏輯不依賴外部因素，因此測試會簡化，而且更容易模擬相依性。

例如，在鬆耦合的架構中，應用程式元件應該能夠在不知道資料存放區詳細資訊的情況下讀取和寫入資料。應用程式元件的責任是將資料提供給 界面（連接埠）。轉接器定義寫入資料存放區的邏輯，可以是資料庫、檔案系統或物件儲存系統，例如 Amazon S3，視應用程式的需求而定。

高層級架構

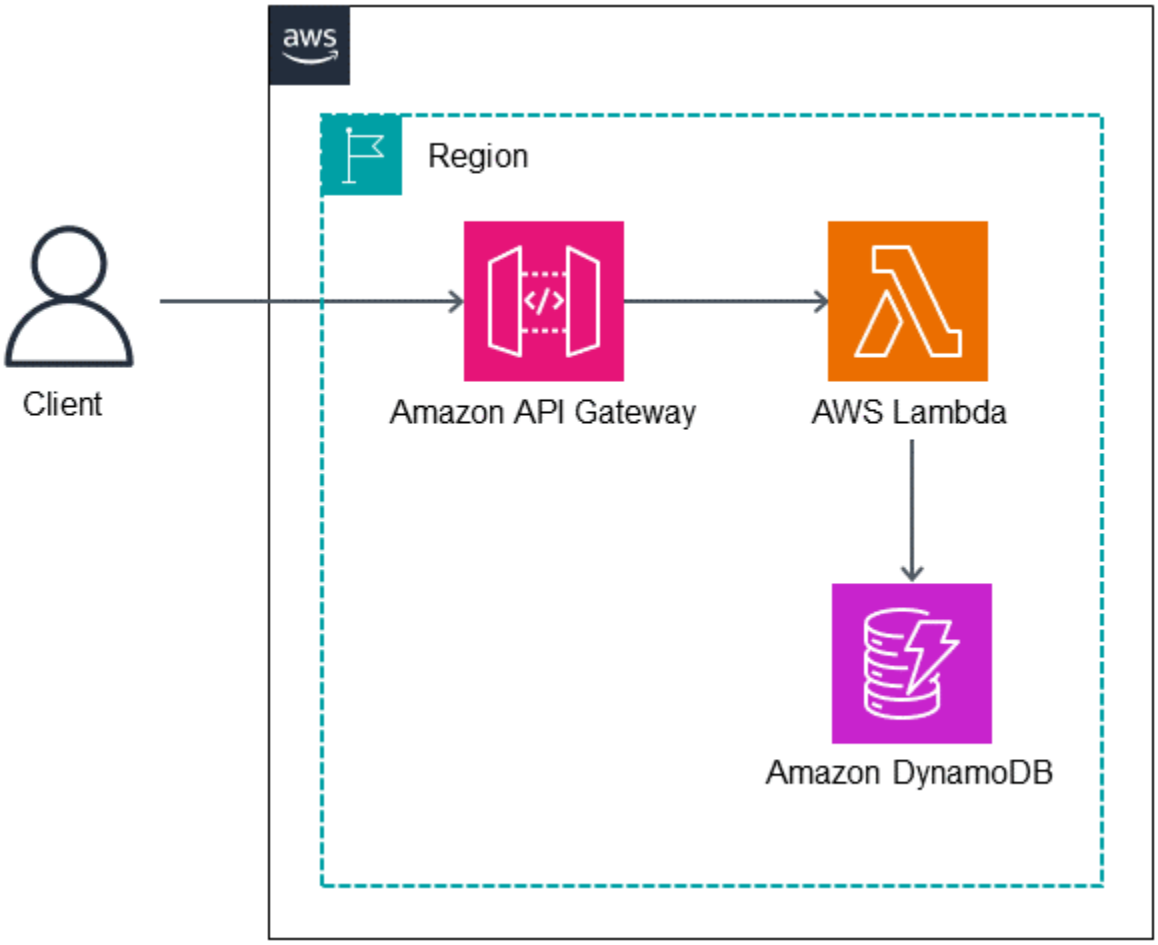
應用程式或應用程式元件包含核心商業邏輯。它從連接埠接收命令或查詢，並透過連接埠將請求傳送至外部演員，這些透過轉接器實作，如下圖所示。



使用 實作 AWS 服務

AWS Lambda 函數通常同時包含商業邏輯和資料庫整合程式碼，這些程式碼會緊密耦合以符合目標。您可以使用六邊形架構模式，將商業邏輯與基礎設施程式碼分開。此區隔可讓單元測試商業邏輯，而不需依賴資料庫程式碼，並改善開發程序的敏捷性。

在下列架構中，Lambda 函數會實作六邊形架構模式。Lambda 函數是由 Amazon API Gateway REST API 啟動。函數實作商業邏輯，並將資料寫入 DynamoDB 資料表。



範本程式碼

本節中的範例程式碼說明如何使用 Lambda 實作網域模型、將其與基礎設施程式碼 （例如存取 DynamoDB 的程式碼） 分開，以及實作函數的單元測試。

網域模型

網域模型類別不知道外部元件或相依性，它只會實作商業邏輯。在下列範例中， 類別`Recipient`是網域模型類別，可檢查保留日期中的重疊。

```
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:
    #.....
```

輸入連接埠

RecipientInputPort 類別會連線至收件人類別，並執行網域邏輯。

```
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 轉接器類別

DDBRecipientAdapter 類別實作對 DynamoDB 資料表的存取。

```
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 函數get_recipient_input_port是 RecipientInputPort類別執行個體的工廠。它使用相關的轉接器執行個體建構輸出連接埠類別的執行個體。

```
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
        }),
    }
```

單元測試

您可以插入模擬類別來測試網域模型類別的商業邏輯。下列範例提供網域模型Recipent類別的單元測試。

```
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 儲存庫

如需此模式的範例架構的完整實作，請參閱 GitHub 儲存庫，網址為 https：//https://github.com/aws-samples/aws-lambda-domain-model-sample。

相關內容

- 六角形架構，作者：Alistair Cockburn
- 使用 開發進化架構 AWS Lambda（日文AWS 部落格文章）

影片

以下影片（日文）討論使用 Lambda 函數在實作網域模型時使用六邊形架構。

發布訂閱模式

意圖

發布-訂閱模式 (亦稱為 pub-sub 模式) 是一種消息傳遞模式，可用來將訊息傳送者 (發布者) 與感興趣的接收者 (訂閱用戶) 解耦。此模式會透過稱為訊息代理程式或路由器 (訊息基礎結構) 的中介發行訊息或事件，藉此實作非同步通訊。發布-訂閱模式會卸除郵件傳遞至訊息基礎結構的責任，藉此增加寄件者的可擴展性和回應能力，讓傳送者可以專注於核心訊息處理。

動機

在分散式架構中，系統組件通常需要在系統內發生事件時向其他組件提供資訊。發布-訂閱模式會分隔考量，讓應用程式可以專注於其核心功能，而訊息基礎結構則會處理訊息路由和可靠傳遞等通訊責任。發布-訂閱模式會啟用非同步傳訊功能來解耦發布者和訂閱用戶。發布者也可以在沒有訂閱用戶知識的情況下傳送訊息。

適用性

出現下列情況時，請使用發布-訂閱模式：

- 如果單一訊息具有不同的工作流程，則需要平行處理。
- 不需要向多個訂閱用戶廣播訊息，也不需要來自接收者的即時回應。
- 系統或應用程式可以容忍資料或狀態的最終一致性。
- 應用程式或元件必須與其他可能使用不同語言、通訊協定或平台的應用程式或服務進行通訊。

問題和考量

- 訂閱用戶可用性：發布者不知道訂閱用戶是否正在接聽，而且訂閱用戶可能並未接聽。已發布的訊息本質上是暫時的，如果訂閱用戶無法使用，則可能會導致捨棄。
- 訊息傳遞保證：通常，發布-訂閱模式無法保證訊息傳遞給所有訂閱用戶類型，但某些服務 (例如 Amazon Simple Notification Service (Amazon SNS) 可以僅提供一次交付給某些訂閱用戶子集。
- 存留時間 (TTL)：訊息有生命週期，如果訊息未在此期間內處理，則會過期。請考慮將已發布的訊息新增至佇列，以便它們可以持續存在，並保證處理時間超過 TTL 期間。

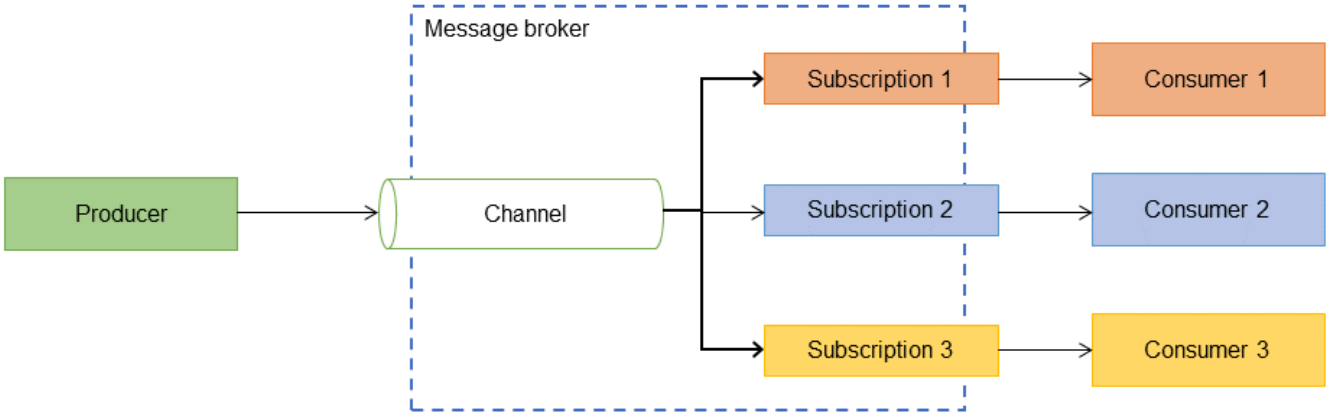
- 訊息相關性：生產者可以設置相關性的時間範圍作為訊息資料的一部分，並且可以在此日期之後丟棄該訊息。在決定如何處理訊息之前，請考慮設計消費者來檢查此資訊。
- 最終一致性：訊息發布到訂閱用戶使用的時間之間存在延遲。這可能會導致在需要強式一致性時，訂閱用戶資料存放區最終變得一致。當生產者和消費者需要接近實時的互動時，最終一致性也可能是一個問題。
- 單向通信：發布-訂閱模式被認為是單向的。如果需要同步回應，則需要具有傳回訂閱通道的雙向傳訊應該考慮使用要求-回覆模式的應用程式。
- 訊息順序：不保證訊息排序。如果消費者需要訂購的訊息，建議您使用 Amazon SNS FIFO 主題來保證訂購。
- 訊息複製：根據訊息基礎結構，重複的訊息可以傳遞給消費者。消費者必須被設計為冪等來處理重複的訊息處理。或者，您也可以使用 Amazon SNS FIFO 主題來保證只能交付一次。
- 訊息篩選：消費者通常只對生產者所發布的訊息子集感興趣。提供機制，讓訂閱用戶透過提供主題或內容篩選器來篩選或縮小收到的訊息範圍。
- 訊息重新顯示：訊息重新顯示功能可能取決於訊息基礎結構。您還可以根據用例提供自定義實現。
- 無效信件佇列：在郵政系統中，無效信件辦事處是處理無法遞送之郵件的設施。在 pub/sub 傳訊功能中，無效字母佇列 (DLQ) 是無法傳遞至已訂閱端點之訊息的佇列。

實作

高層級架構

在發布-訂閱模式中，稱為訊息代理程式或路由器的非同步訊息子系統會追蹤訂閱。當生產者發布事件時，訊息基礎結構會傳送訊息給每個消費者。將訊息傳送給訂閱用戶之後，就會從訊息基礎結構中移除，因此無法重新顯示訊息，而新訂閱用戶也看不到該事件。訊息代理程式或路由器會透過下列方式來解耦事件生產者與訊息消費者：

- 為生產者提供輸入通道，以使用定義的訊息格式來發布封裝成訊息的事件。
- 為每個訂閱建立個別的輸出通道。訂閱是取用者的連線，它們透過此連線接聽與特定輸入通道相關聯的事件訊息。
- 發布事件時，將訊息從輸入通道複製到所有消費者的輸出通道。



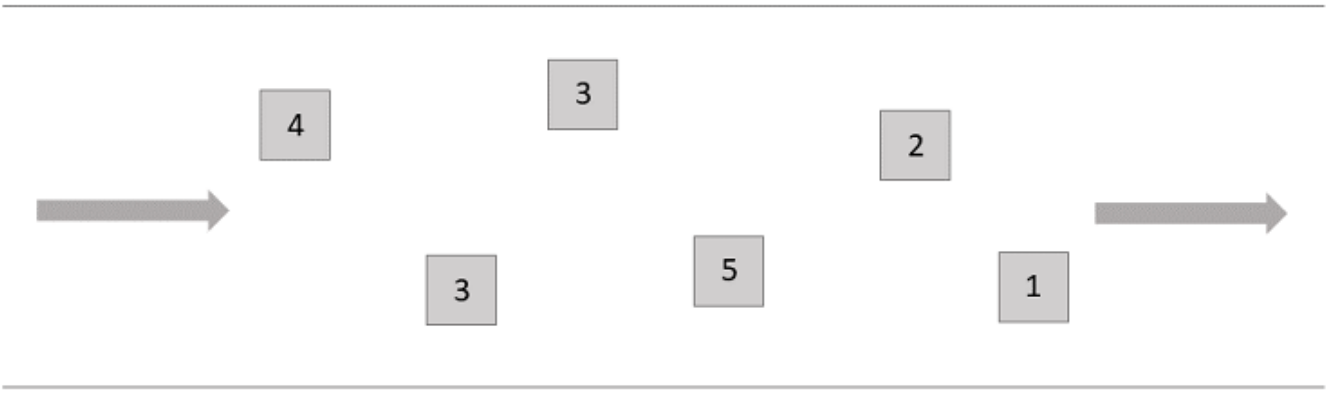
使用 AWS 服務來實作

Amazon SNS

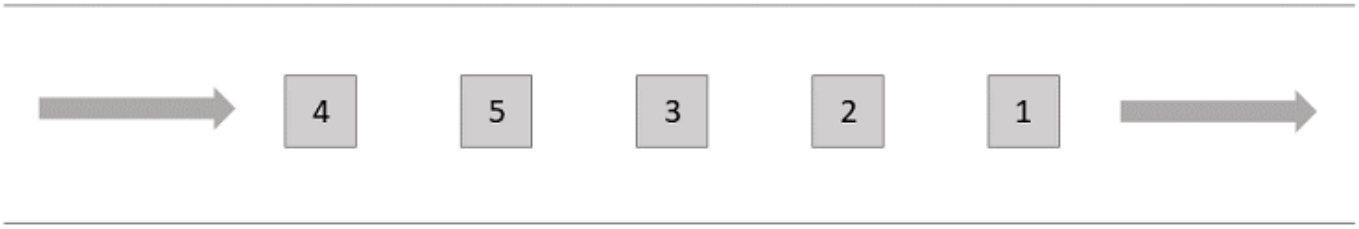
Amazon SNS 是一種全受管的發布者-訂閱用戶服務，可提供應用程式至應用程式 (A2A) 傳訊功能以解耦分散式應用程式。它還提供應用程式至人員 (A2P) 訊息，用於傳送簡訊、電子郵件和其他推播通知。

Amazon SNS 提供兩種類型的主題：標準主題和先進先出 (FIFO)。

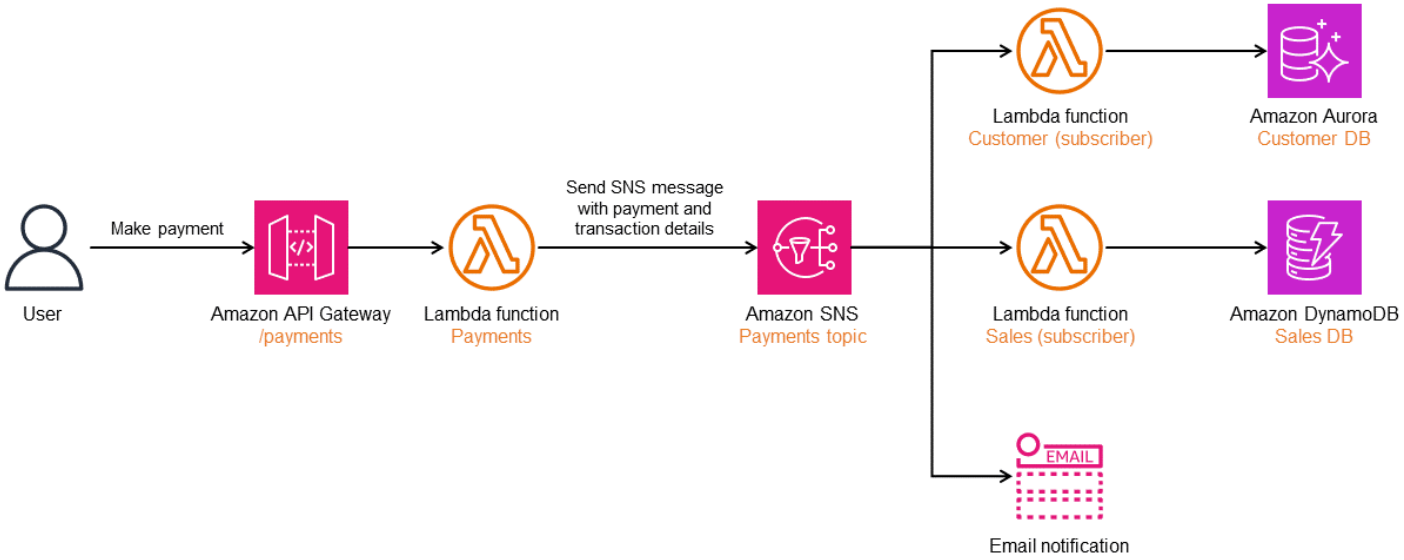
- 標準主題支援每秒不限數量的訊息，並提供最佳的排序和重複資料刪除功能。



- FIFO 主題提供嚴格的排序和重複資料刪除功能，每個 FIFO 主題最多支援每秒 300 則訊息或每秒 10 MB (以先到者為準)。

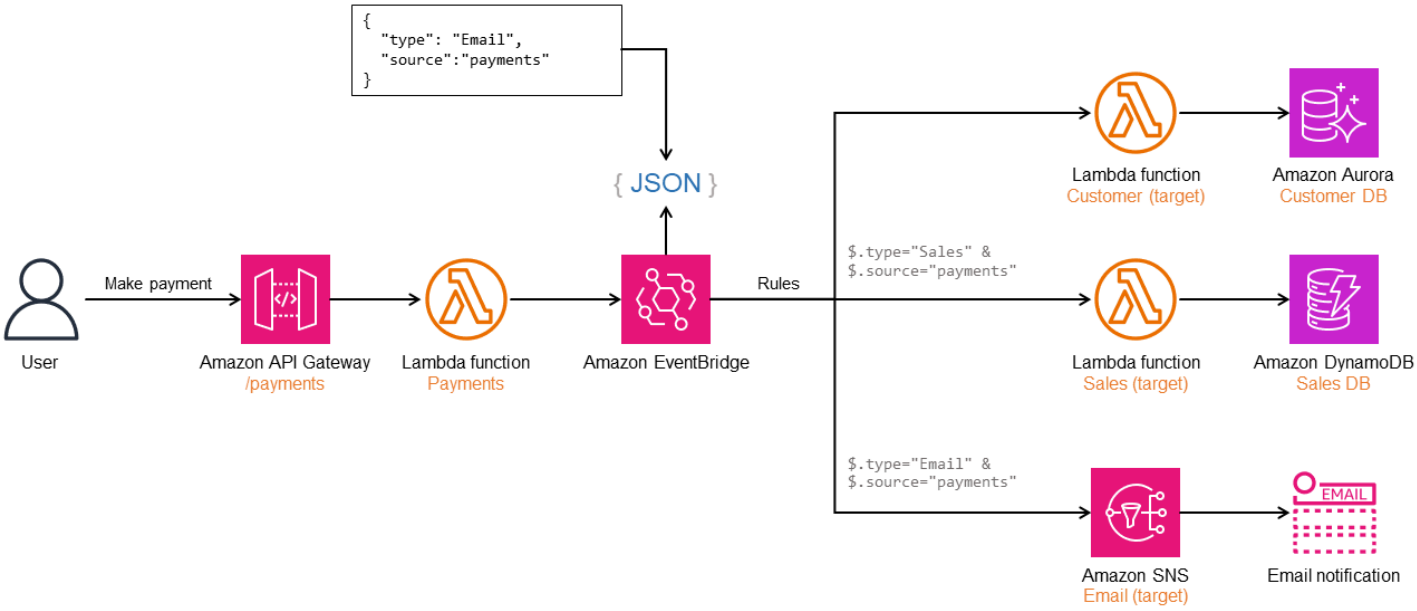


下圖顯示如何使用 Amazon SNS 實作發布-訂閱模式。使用者付款後，Payments Lambda 函數會將 SNS 訊息傳送至 Payments SNS 主題。此 SNS 主題有三個訂閱用戶。每個訂閱用戶都會收到訊息的副本並加以處理。



Amazon EventBridge

當您需要更複雜地將來自多個生產者的訊息跨不同協定路由傳送至訂閱的消費者，或直接和散發訂閱時，可以使用 Amazon EventBridge。EventBridge 也支援以內容為基礎的路由、篩選、排序以及分割或彙總。在下圖中，EventBridge 是用來建立發布-訂閱模式的版本，其中訂閱用戶是使用事件規則來定義。使用者付款後，Payments Lambda 函數會根據具有三個指向不同目標的自訂結構描述，使用預設事件匯流排將訊息傳送至 EventBridge。每個微服務都會處理訊息並執行必要的動作。



研討會

- 在 AWS 上建立事件驅動架構
- 使用 Amazon Simple Queue Service (Amazon SQS) 和 Amazon Simple Notification Service (Amazon SNS) 來傳送扇出事件通知

部落格參考

- 選擇無伺服器應用程式的傳訊服務
- 使用適用於 Amazon SNS、Amazon SQS 和 AWS Lambda 的 DLQ 來設計耐用的無伺服器應用程式
- 使用 Amazon SNS 傳訊篩選簡化 pub/sub 傳訊

相關內容

- pub/sub 傳訊的功能

使用輪詢重試模試

意圖

使用退避模式的重試透過透明重試因暫時性錯誤而失敗的操作來改善應用程式穩定性。

動機

在分散式架構中，暫時性錯誤可能是由於服務限流、暫時失去網路連線或暫時服務無法使用所造成。由於這些暫時性錯誤而失敗的自動重試操作可改善使用者體驗和應用程式彈性。不過，頻繁重試可能會使網路頻寬過載並導致爭用。指數退避是一種技術，透過增加指定重試次數的等待時間來重試操作。

適用性

在下列情況下，使用重試與退避模式：

- 您的服務經常調節請求以防止過載，導致呼叫程序發生 429 個請求例外狀況。
- 網路是分散式架構中看不見的參與者，而暫時性網路問題會導致失敗。
- 正在呼叫的服務暫時無法使用，導致失敗。除非您使用此模式引入退避逾時，否則頻繁重試可能會導致服務降級。

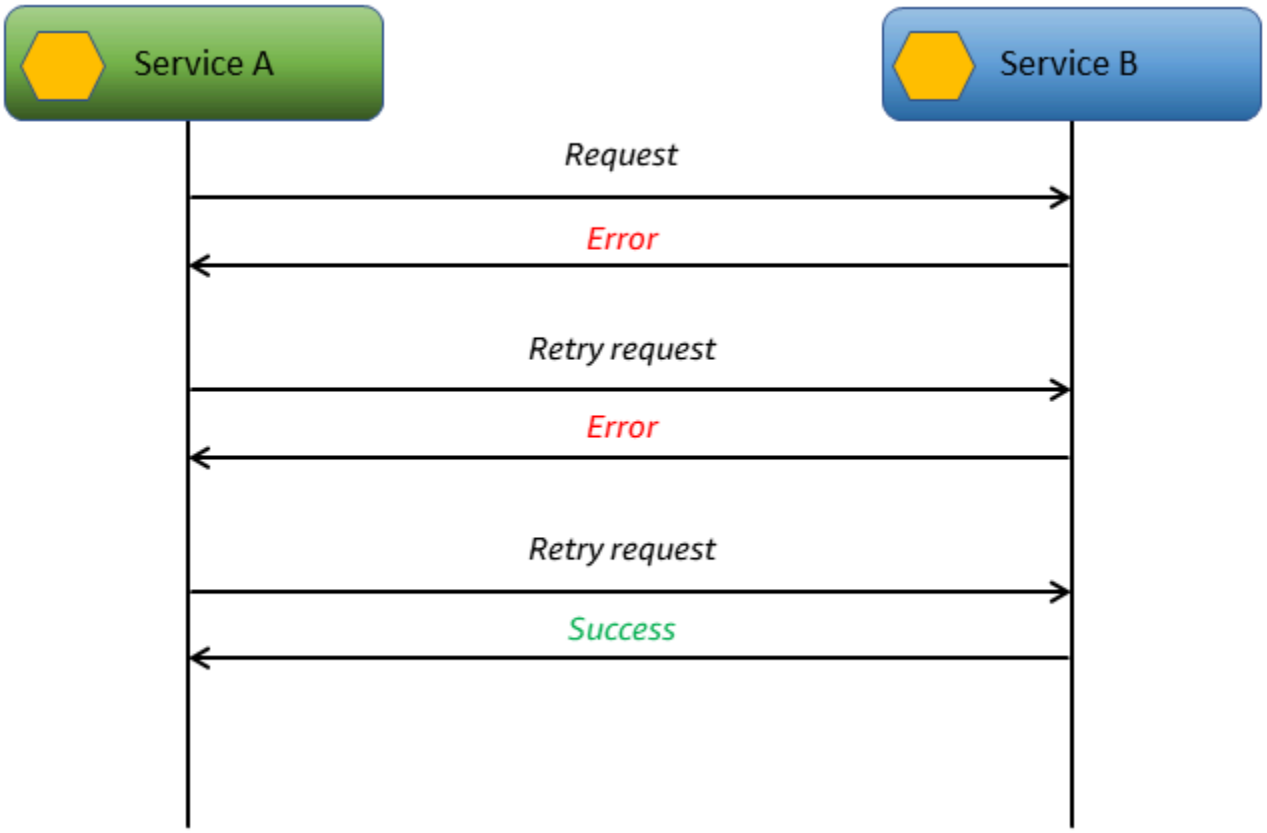
問題和考量

- 冪等性：如果對 方法的多個呼叫與系統狀態的單一呼叫具有相同的效果，則操作會被視為冪等性。當您使用具有退避模式的重試時，操作應該是等冪的。否則，部分更新可能會損毀系統狀態。
- 網路頻寬：如果太多次重試佔用網路頻寬，會導致回應時間變慢，則可能會發生服務降級。
- 快速失敗案例：對於非暫時性錯誤，如果您可以判斷失敗的原因，則使用斷路器模式快速失敗會更有效率。
- 退避率：引入指數退避可能會影響服務逾時，導致最終使用者的等待時間較長。

實作

高層級架構

下圖說明 Service A 如何重試對 Service B 的呼叫，直到成功傳回回應為止。如果服務 B 在嘗試幾次後仍未傳回成功回應，服務 A 可以停止重試並將失敗傳回給其發起人。

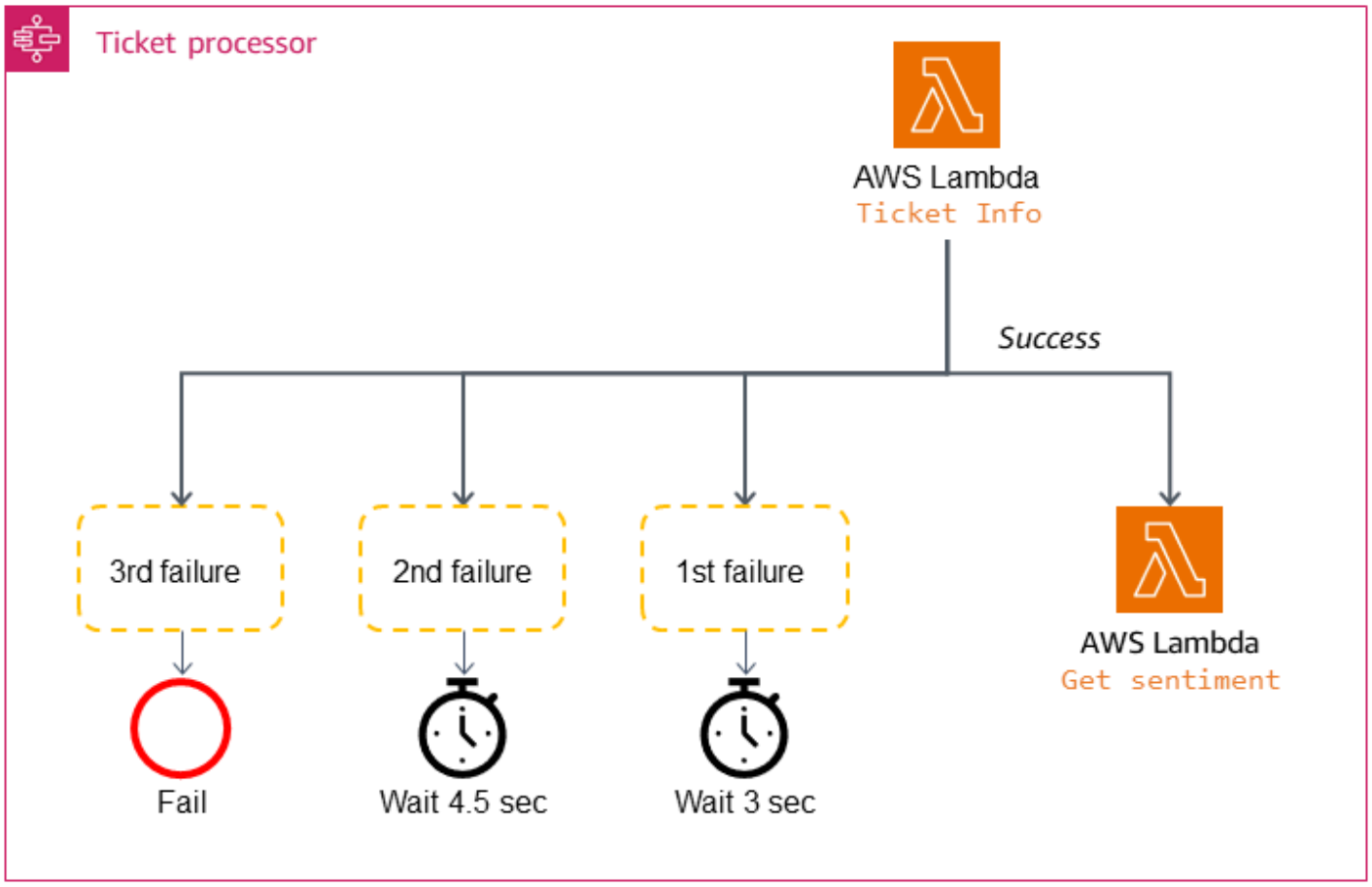


使用 AWS 服務實作

下圖顯示客戶支援平台上的票證處理工作流程。來自不滿意客戶的票證會透過自動呈報票證優先順序來加速。`Ticket info` Lambda 函數會擷取票證詳細資訊並呼叫 `Get sentiment` Lambda 函數。`Get sentiment` Lambda 函數透過將描述傳遞至 Amazon Comprehend （未顯示）來檢查客戶情緒。

如果呼叫 `Get sentiment` Lambda 函數失敗，工作流程會重試操作三次。可讓您設定退避值，以 AWS Step Functions 允許指數退避。

在此範例中，最多會設定三次重試，增加乘數為 1.5 秒。如果第一次重試發生在 3 秒之後，第二次重試發生在 3 x 1.5 秒 = 4.5 秒之後，而第三次重試發生在 4.5 x 1.5 秒 = 6.75 秒之後。如果第三個重試失敗，工作流程會失敗。退避邏輯不需要任何自訂程式碼，由 提供做為組態 AWS Step Functions。



範本程式碼

下列程式碼顯示使用退避模式進行重試的實作。

```
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 儲存庫

如需此模式範例架構的完整實作，請參閱 GitHub 儲存庫，網址為 https：//https://github.com/aws-samples/retry-with-backoff。

相關內容

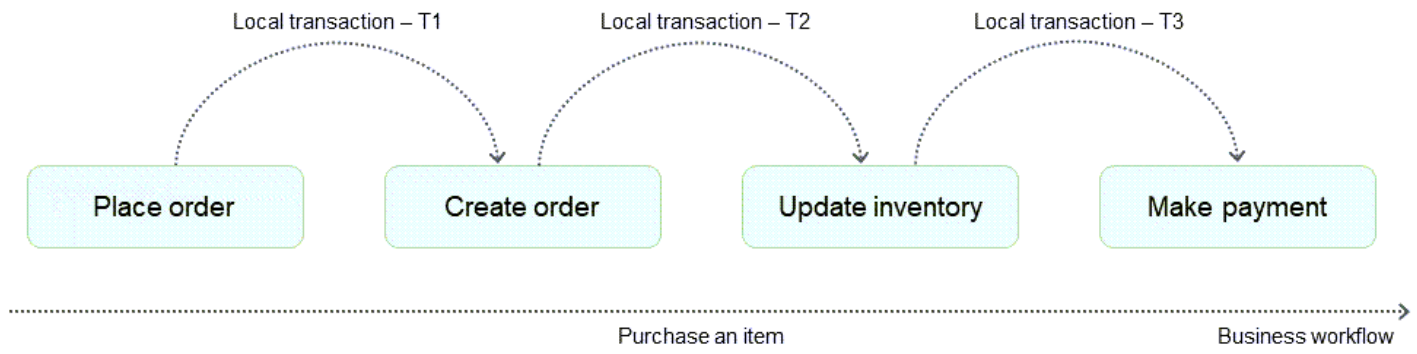
- 具有抖動的逾時、重試和退避 (Amazon Builders' Library)

系列事件模式

系列事件是由一系列本機交易組成。系列事件中的每個本地端交易都會更新資料庫並觸發下一個本地端交易。如果一個交易失敗，系列事件執行補償交易，以恢復由先前的交易所做的資料庫更改。

這一序列的本地端交易有助於透過使用延續和補償原則達成業務工作流程。延續原則決定工作流程的向前復原，而補償原則決定向後復原。如果更新在交易中的任何步驟失敗，則系列事件會針對繼續 (重試交易) 或補償 (回到先前的資料狀態) 發布事件。這樣可確保資料完整性得以維持，並在整個資料存放區中保持一致。

例如，當使用者向線上零售商購買帳本時，處理包含代表業務工作流程的一系列異動，例如訂單建立、存貨更新、付款及出貨。為了完成此工作流程，分散式架構會發出一系列本機交易，以便在訂單資料庫中建立訂單、更新庫存資料庫，以及更新付款資料庫。當處理成功時，系統會依序呼叫這些交易以完成業務工作流程，如下列圖表所示。但是，如果這些本機交易中的任何一項失敗，系統應該能夠決定適當的下一個步驟 — 也就是向前復原或向後復原。



下列兩種案例有助於判斷下一個步驟是向前復原還是向後復原：

- 平台層級失敗，其中基礎結構出了問題，並導致交易失敗。在這種情況下，系列事件模式可以透過重試本地交易並繼續業務流程來執行向前恢復。
- 應用程式層級失敗，即付款服務因為付款無效而失敗。在這種情況下，系列事件模式可以透過發出補償性交易來更新庫存和訂單資料庫，並恢復其先前的狀態來執行向後恢復。

系列事件模式處理業務工作流程，並確保透過向前恢復達到理想的終止狀態。在失敗的情況下，它透過使用向後恢復，以避免資料一致性問題恢復本地端交易。

系列事件模式有兩種變體：編排和協同運作。

系列事件編排

系列事件編排模式取決於微服務發布的事件。系列事件參與者 (微服務) 訂閱事件並根據事件觸發器採取行動。例如，下列圖表中的訂單服務會發出 `OrderPlaced` 事件。存貨服務會訂閱該事件，並在發出 `OrderPlaced` 事件時更新庫存。同樣地，參與者服務會根據發出事件的內容執行動作。

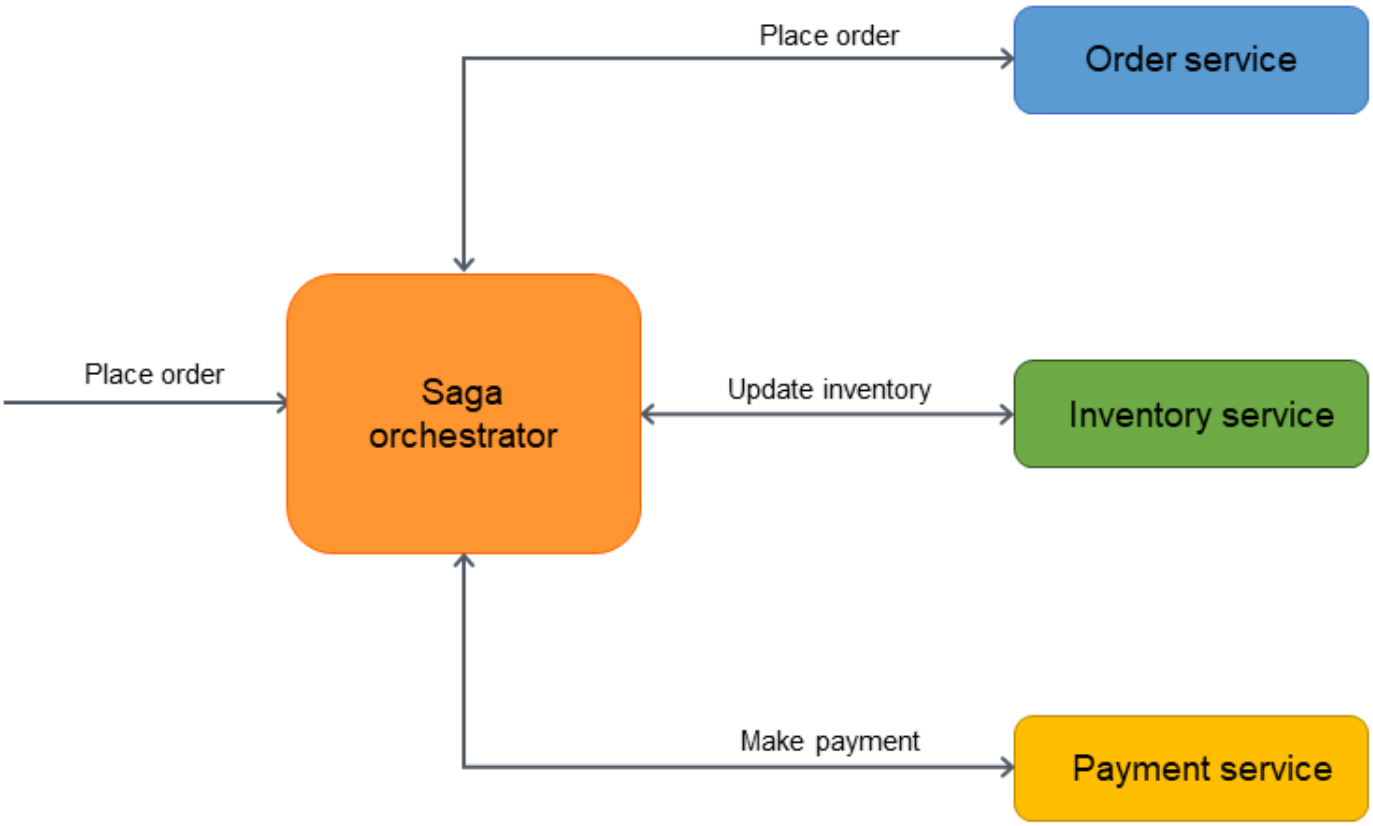
當系列事件中只有少數參與者，且您需要一個沒有單點故障的簡單實作時，系列事件編排模式非常適合。當加入了更多參與者時，使用此模式來追蹤參與者之間的相依性會變得更加困難。

如需詳細評論，請參閱本指南的系列事件編排區段。

系列事件協同運作

系列事件協同運作模式有個稱為協調器的中央協調器。系列事件協調器管理和協調整個交易生命週期。它知道要執行以完成交易的一系列步驟。為了執行一個步驟，它發送一條訊息到參與者微服務來執行操作。參與者微服務完成操作，並將訊息發送回協調器。根據收到的訊息，協調器決定要在交易中接下來執行哪些微服務。

當有許多參與者時，系列事件協同運作模式是合適的，且系列事件參與者之間需要鬆耦合。協調器透過使參與者鬆耦合封裝邏輯中的複雜性。但是，協調器可能會成為單一失敗點，因為它控制了整個工作流程。



如需詳細評論，請參閱本指南的系列事件協同運作區段。

系列事件編排模式

意圖

系列事件編排模式有助於使用事件訂閱跨越多個服務的分散式交易中保持資料完整性。在分散式交易中，交易完成前可以呼叫多個服務。當服務將資料儲存在不同的資料存放區時，在這些資料存放區之間維持資料一致性可能會具有挑戰性。

動機

交易是可能涉及多個步驟的單一工作單元，除非完整執行所有步驟，否則不執行任何步驟，藉此使得資料存放區保持其一致的狀態。原子性、一致性、隔離性和耐久性 (ACID) 等詞彙定義了交易的屬性。關聯式資料庫提供 ACID 交易來維持資料一致性。

為了維持交易的一致性，關聯式資料庫使用兩階段遞交 (2PC) 方法。這種方法是由準備階段和遞交階段組成。

- 在準備階段，協調處理會要求交易的參與處理 (參與者) 承諾遞交或回復交易。
- 在遞交階段，協調處理會要求參與者遞交該筆交易。如果參與者無法同意在準備階段遞交，則系統會回復交易。

在遵循 database-per-service 設計模式的分散式系統中，系統不允許兩階段遞交。這是因為每筆交易均分散在不同的資料庫中，而且沒有單一控制器可以協調與關聯式資料存放區中的兩階段遞交類似的處理。在這種情況下，其中一種解決方案是使用系列事件編排模式。

適用性

在以下情況下使用系列事件編排模式：

- 在橫跨多個資料存放區的分散式交易中，您的系統需要資料完整性和一致性。
- 資料存放區 (例如 NoSQL 資料庫) 不提供 2PC 來提供 ACID 交易，您需要在單一交易中更新多個資料表，而在應用程式邊界內實作 2PC 將是一項複雜的工作。
- 管理參與者交易的集中控制程序可能會成為單一失敗點。
- 系列事件參與者是獨立的服務，需要鬆耦合。
- 企業領域中有邊界的前後關聯之間存在通訊。

問題和考量

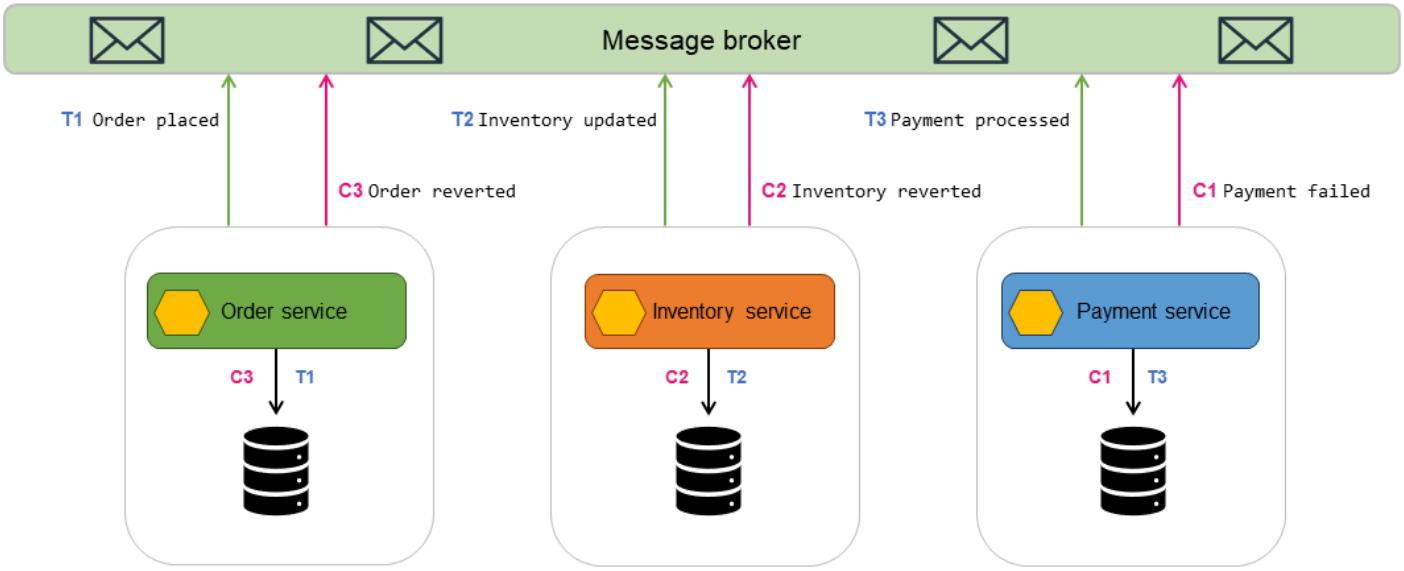
- 複雜性：隨著微服務數量的增加，由於微服務之間的交互數量，系列事件編排可能變得難以管理。此外，補償性交易和重試會增加應用程式程式碼的複雜性，這可能會導致維護上的額外負荷。當系列事件中只有少數參與者，且您需要一個沒有單點故障的簡單實作時，編排非常適合。當加入了更多參與者時，使用此模式來追蹤參與者之間的相依性會變得更加困難。
- 彈性實作：在系列事件編排中，與系列事件協同運作相比，在全域範圍內實施超時、重試和其他彈性模式會更加困難。編排必須在個別元件上執行，而不是在協調器層級執行。
- 循環依賴關係：參與者消耗彼此發布的訊息。這可能會導致循環依賴性，導致代碼複雜性和維護開銷，以及可能的死結。
- 雙寫入問題：微服務必須以原子方式更新資料庫並發布事件。任一作業失敗都可能導致不一致的狀態。解決這個問題的其中一種方法，是使用交易寄件匣模式。
- 保存事件：系列事件參與者根據發布的事件採取行動。請務必依照事件發生的順序儲存事件，以供稽核、偵錯和重新執行之用。如果需要重新執行系統狀態才能還原資料一致性，您可以使用事件來源模式將事件保留在事件存放區中。事件存放區也可用於稽核和疑難排解，因為它們會反映系統中的每一項變更。

- 最終一致性：本地端交易的序列處理會導致最終一致性，此情況對於需要強式一致性的系統來說可能是個挑戰。您可以設定業務團隊對一致性模式的期望來解決此問題，或重新評估使用案例並切換到提供強式一致性的資料庫。
- 等冪性：系列事件參與者必須是等冪的，以便在由意外當機和協調器故障引起的暫時性故障時允許重複執行。
- 交易隔離：系列事件模式缺少交易隔離，這是 ACID 交易中的四個屬性之一。交易的隔離程度決定了其他並行交易能影響該筆交易在資料上操作的程度多寡。交易的並行協同運作可能會導致過時資料。我們建議使用語義鎖定來處理此類情況。
- 可觀測性：可觀測性是指詳細的日誌記錄和追蹤，藉此疑難排解實作和協同運作中的問題。當系列事件參與者的數量增加時，這會變得很重要，導致偵錯的複雜性。與系列事件協同運作相比，在系列事件編排中更難達成端到端監控和報告。
- 延遲問題：當系列事件是由數個步驟組成時，補償性交易可能會增加整體回應時間的延遲。如果交易進行同步呼叫，這可能會進一步增加延遲。

實作

高層級架構

在下面的架構圖中，系列事件編排有三個參與者：訂單服務、庫存服務和支付服務。完成交易需要三個步驟：T1、T2 和 T3。三種補償性交易會將資料恢復到初始狀態：C1、C2 和 C3。



- 訂單服務會執行本機交易 T1，以原子方式更新資料庫，並將 `Order placed` 訊息發布至訊息代理程式。

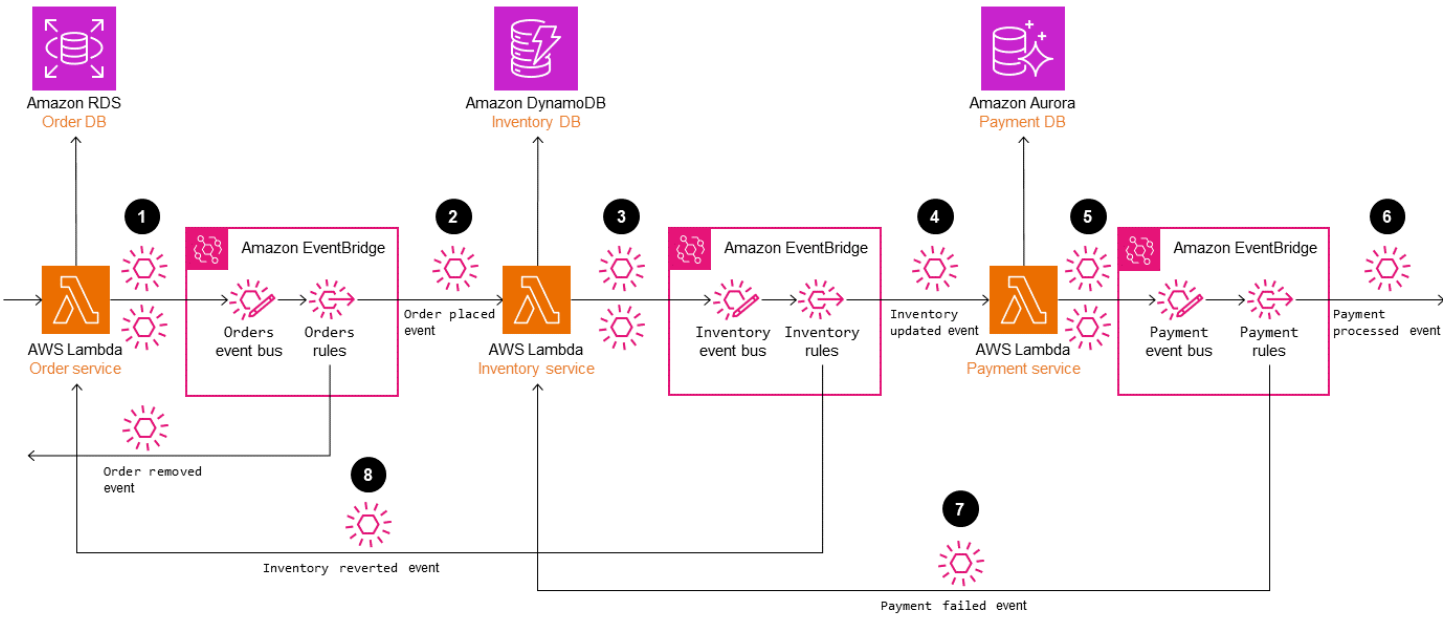
- 庫存服務會訂閱訂單服務訊息，並接收訂單已建立的訊息。
- 庫存服務會執行本機交易 T2，以原子方式更新資料庫，並將 `Inventory updated` 訊息發布至訊息代理程式。
- 付款服務會訂閱來自庫存服務的訊息，並收到清查已更新的訊息。
- 付款服務會執行本機交易 T3，以原子方式使用付款詳細資訊來更新資料庫，並將 `Payment processed` 訊息發布至訊息代理程式。
- 如果付款失敗，付款服務會執行補償性交易 C1，該交易會以原子方式回復資料庫中的付款，並將 `Payment failed` 訊息發布給訊息代理程式。
- 執行補償交易 C2 和 C3 以還原資料一致性。

使用 AWS 服務來實作

您可以透過使用 Amazon EventBridge 來實現系列事件編排模式。EventBridge 使用事件來連接應用程式元件。它透過事件總線或管道處理事件。事件匯流排是接收事件並將事件傳遞至零個或多個目的地或目標的一種路由器。與事件匯流排相關聯的規則會在事件到達時評估事件，並將事件傳送至目標進行處理。

在以下體系結構中：

- 微型服務 (訂單服務、庫存服務和付款服務) 是以 Lambda 函數的形式實作。
- 有三種自訂 EventBridge 匯流排：`Orders` 事件匯流排、`Inventory` 事件匯流排和 `Payment` 事件匯流排。
- `Orders` 規則、`Inventory` 規則和 `Payment` 規則會與傳送至對應事件匯流排的事件配對，並調用 Lambda 函數。



在成功的情況下，當下訂單時：

1. 訂單服務會處理要求，並將事件傳送至 Orders 事件匯流排。
2. Orders 規則符合事件並啟動庫存服務。
3. 庫存服務會更新庫存，並將事件傳送至 Inventory 事件匯流排。
4. Inventory 規則符合事件並啟動付款服務。
5. 付款服務會處理付款，並將事件傳送至 Payment 事件匯流排。
6. 這些 Payment 規則會與事件相符，並將 Payment processed 事件通知傳送給接聽程式。

   或者，當訂單處理發生問題時，EventBridge 規則會啟動補償性交易，以還原資料更新以維持資料一致性和完整性。
7. 如果付款失敗，Payment 規則會處理事件並啟動庫存服務。存貨服務會執行補償性異動，以回復存貨。
8. 還原庫存後，庫存服務會將 Inventory reverted 事件傳送至 Inventory 事件匯流排。此事件由 Inventory 規則處理。它會啟動訂單服務，該服務會執行補償性交易以移除訂單。

相關內容

- 系列事件協同運作模式
- 交易寄件匣模式

- 使用輪詢重試模試

系列事件協同運作模式

意圖

系列事件協同運作模式使用中央協調器 (協調器) 來協助維持橫跨多個服務的分散式交易中的資料完整性。在分散式交易中，交易完成前可以呼叫多個服務。當服務將資料儲存在不同的資料存放區時，在這些資料存放區之間維持資料一致性可能會具有挑戰性。

動機

交易是可能涉及多個步驟的單一工作單元，除非完整執行所有步驟，否則不執行任何步驟，藉此使得資料存放區保持其一致的狀態。原子性、一致性、隔離性和耐久性 (ACID) 等詞彙定義了交易的屬性。關聯式資料庫提供 ACID 交易來維持資料一致性。

為了維持交易的一致性，關聯式資料庫使用兩階段遞交 (2PC) 方法。這種方法是由準備階段和遞交階段組成。

- 在準備階段，協調處理會要求交易的參與處理 (參與者) 承諾遞交或回復交易。
- 在遞交階段，協調處理會要求參與者遞交該筆交易。如果參與者無法同意在準備階段遞交，則系統會回復交易。

在遵循 database-per-service 設計模式的分散式系統中，系統不允許兩階段遞交。這是因為每筆交易均分散在不同的資料庫中，而且沒有單一控制器可以協調與關聯式資料存放區中的兩階段遞交類似的處理。在這種情況下，其中一種解決方案是使用系列事件協同運作模式。

適用性

出現下列情況時，請使用系列事件協同運作模式：

- 在橫跨多個資料存放區的分散式交易中，您的系統需要資料完整性和一致性。
- 資料存儲不提供 2PC 來提供 ACID 交易，並且在應用程式邊界內實作 2PC 是一項複雜的任務。
- 您有不提供 ACID 交易的 NoSQL 資料庫，並且需要在單個交易中更新多個資料表。

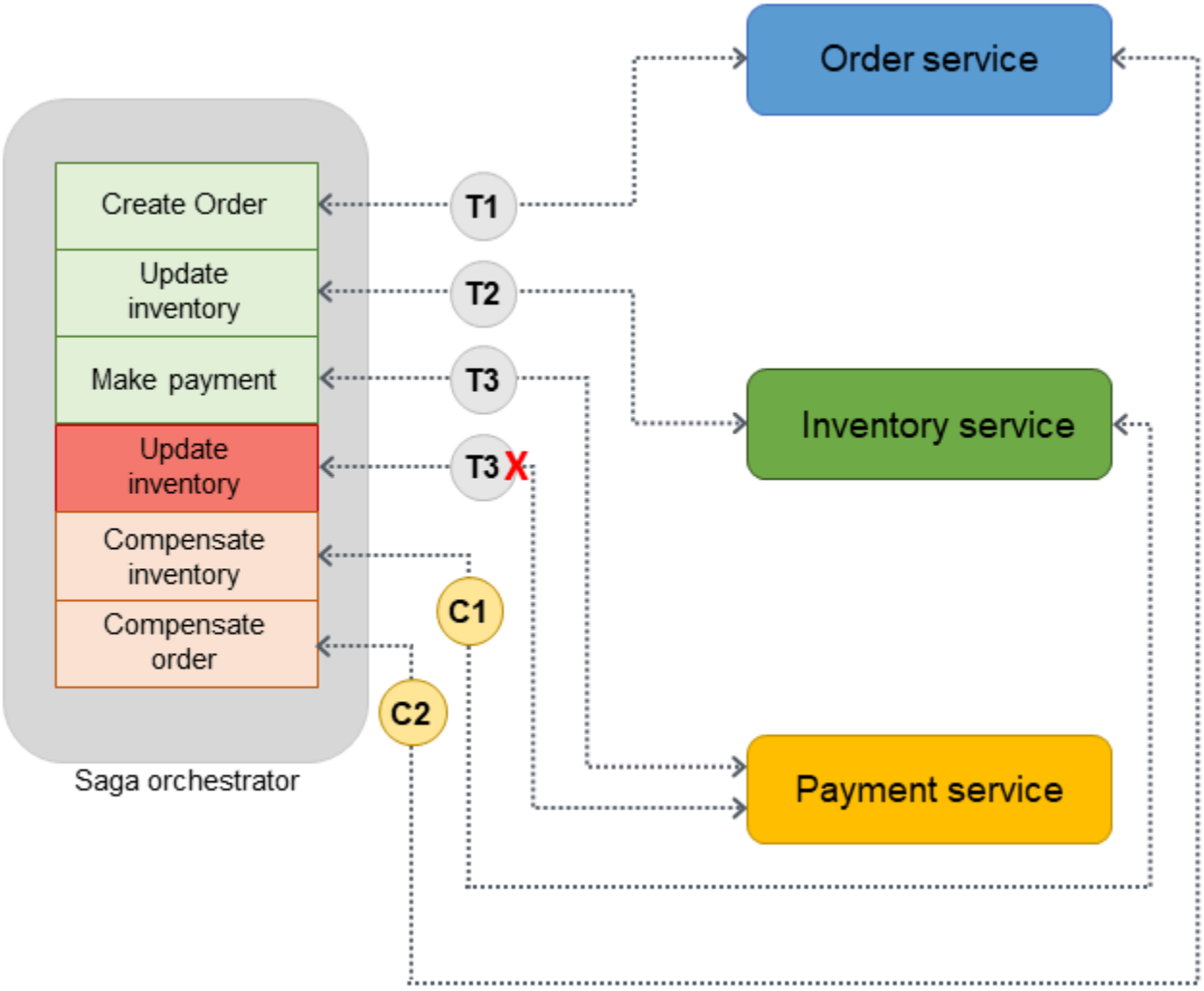
問題和考量

- 複雜性：補償性交易和重試會增加應用程式程式碼的複雜性，這可能會導致維護額外負荷。
- 最終一致性：本地端交易的序列處理會導致最終一致性，此情況對於需要強式一致性的系統來說可能是個挑戰。您可以設定業務團隊對一致性模式的期望來解決此問題，或透過切換到提供強式一致性的資料存放區的方式來解決此問題。
- 等冪性：系列事件參與者必須是等冪的，以便在由意外當機和協調器故障引起的暫時性故障時允許重複執行。
- 交易隔離：系列事件缺乏交易隔離。交易的並行協同運作可能會導致過時資料。我們建議使用語義鎖定來處理此類情況。
- 可觀測性：可觀測性是指詳細的日誌記錄和追蹤，藉此疑難排解執行和協同運作中的問題。當系列事件參與者的數量增加時，這會變得很重要，導致偵錯的複雜性。
- 延遲問題：當系列事件是由數個步驟組成時，補償性交易可能會增加整體回應時間的延遲。在這種情況下避免同步呼叫。
- 單點失敗：協調器可能會因為協調整個交易而成為單一失敗點。在某些情況下，由於這個問題，系列事件的編排模式是首選。

實作

高層級架構

在下面的架構圖中，系列事件協調器有三個參與者：訂單服務、庫存服務和支付服務。完成交易需要三個步驟：T1、T2 和 T3。系列事件協調器知道的步驟，並按照所需的順序執行它們。當步驟 T3 失敗 (付款失敗) ，協調器執行補償交易 C1 和 C2 將資料恢復到初始狀態。



當交易分散在多個資料庫中時，您可以使用 AWS Step Functions 來實作系列事件協同運作。

使用 AWS 服務來實作

範例解決方案會使用 Step Functions 中的標準工作流程來實作系列事件協同運作模式。

當客戶呼叫 API 時，會調用 Lambda 函數，而預先處理會在 Lambda 函數中進行。函數會啟動 Step Functions 工作流程，以開始處理分散式交易。如果不需要預先處理，您可以直接從 API Gateway 啟動 Step Functions 工作流程，而無需使用 Lambda 函數。

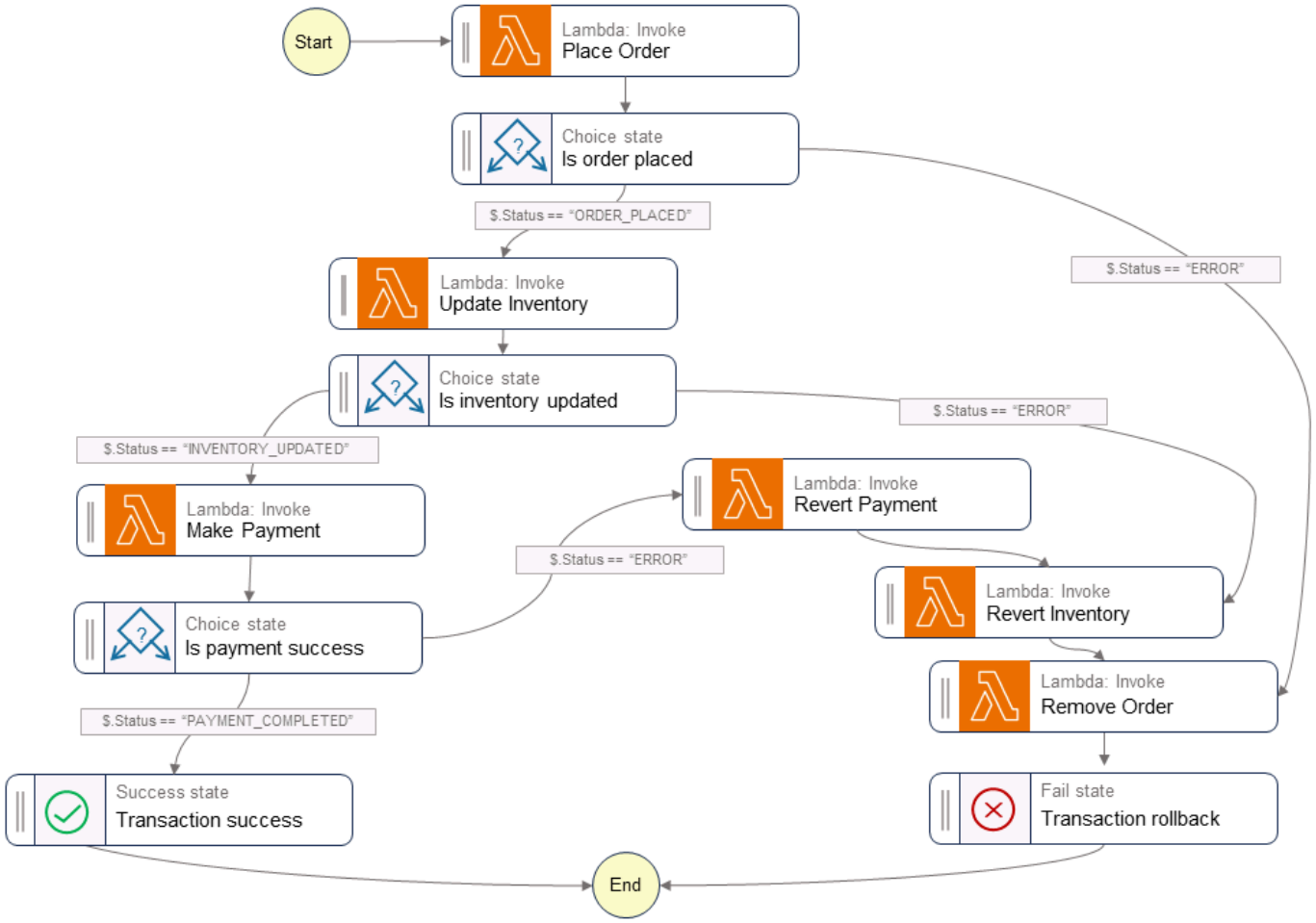
使用 Step Functions 可減輕以往實作系列事件協同運作模式時會出現的單點故障問題。Step Functions 具有內建的容錯能力，並維護每個 AWS 區域中多個可用區域的服務容量，以保護應用程式免受個別機器或資料中心故障的影響。這有助於確保服務本身及其操作的應用程式工作流程的高可用性。

Step Functions 工作流程

Step Functions 狀態機器可讓您針對模式實作設定以決策為基礎的控制流程需求。Step Functions 工作流程會呼叫個別服務，以進行訂單放置、存貨更新及付款處理，以完成異動，並傳送事件通知以供進一步處理。Step Functions 工作流程充當協調器來協調交易。如果工作流程包含任何錯誤，協調器會執行補償性交易，以確保跨服務維護資料完整性。

下列圖表顯示在 Step Functions 工作流程內執行的步驟。`Place Order`、`Update Inventory` 和 `Make Payment` 步驟表示成功路徑。訂單已下訂單，更新庫存，並在 `Success` 狀態傳回呼叫者之前處理付款。

`Revert Payment`、`Revert Inventory` 和 `Remove Order` Lambda 函數會指出當工作流程中的任何步驟失敗時，協調器執行的補償性交易。如果工作流程在 `Update Inventory` 步驟中失敗，協調器會先呼叫 `Revert Inventory` 和 `Remove Order` 步驟，再將 `Fail` 狀態傳回呼叫者。這些補償交易可確保維護資料的完整性。庫存會退回其原始層次，並回復訂單。



範本程式碼

下列範例程式碼顯示如何使用 Step Functions 建立系列事件協調器。若要檢視完整的程式碼，請參閱 GitHub 儲存庫以取得此範例。

任務定義

```
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);
```

步進函數和狀態機器定義

```
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 儲存庫

如需此模式範例架構的完整實作，請參閱 GitHub 存放庫，網址為 https://github.com/aws-samples/saga-orchestration-netcore-blog。

部落格參考

- 使用系列事件協同運作模式建置無伺服器分散式應用程式

相關內容

- 系列事件編排模式
- 交易寄件匣模式

影片

下列影片討論如何使用 實作 saga 協同運作模式 AWS Step Functions。

散佈集合模式

意圖

散佈收集模式是一種訊息路由模式，涉及將類似或相關的請求廣播給多個收件人，並使用稱為彙總工具的元件將其回應彙總回單一訊息。此模式有助於實現平行化、減少處理延遲，並處理非同步通訊。使用同步方法實作散佈收集模式很簡單，但更強大的方法涉及在非同步通訊中將其實作為訊息路由，無論是否使用訊息服務。

動機

在應用程式處理中，可能需要很長時間才能循序處理的請求可以分割成多個平行處理的請求。您也可以透過 API 呼叫將請求傳送至多個外部系統，以取得回應。當您需要來自多個來源的輸入時，散佈集模式很有用。Scatter-gather 會彙總結果，以協助您做出明智的決定，或選取請求的最佳回應。

散佈集模式包含兩個階段，如其名稱所示：

- 散佈階段會處理請求訊息，並將其平行傳送給多個收件人。在此階段，應用程式會將請求分散到整個網路，並繼續執行，而無需等待立即回應。
- 在收集階段期間，應用程式會收集收件人的回應，並將其篩選或合併為統一的回應。收集所有回應後，它們可以彙總為單一回應，也可以選擇最佳回應進行進一步處理。

適用性

在下列情況下使用散佈集模式：

- 您計劃從各種 APIs彙總和合併資料，以建立準確的回應。模式會將來自不同來源的資訊合併為一個有凝聚力的整個。例如，預訂系統可以向多個收件人提出請求，從多個外部合作夥伴取得引號。
- 相同的請求必須同時傳送給多個收件人，才能完成交易。例如，您可以使用此模式平行查詢庫存資料，以檢查產品的可用性。
- 您想要實作可靠且可擴展的系統，透過將請求分散到多個收件人來實現負載平衡。如果某個收件人失敗或遇到高負載，其他收件人仍然可以處理請求。
- 您想要在實作涉及多個資料來源的複雜查詢時最佳化效能。您可以將查詢分散至相關資料庫、收集部分結果，並將其合併為完整的答案。

- 您正在實作一種映射縮減處理，其中資料請求會路由到多個資料處理端點以進行分片和複寫。部分結果會經過篩選並合併，以構成正確的回應。
- 您想要在金鑰值資料庫中的大量寫入工作負載中，將寫入操作分散到分割區金鑰空間。彙總工具會透過查詢每個碎片中的資料來讀取結果，然後將它們合併為單一回應。

問題和考量

- 容錯能力：此模式依賴平行工作的多個收件人，因此正常處理失敗至關重要。若要減輕收件人失敗對整體系統的影響，您可以實作備援、複寫和故障偵測等策略。
- 橫向擴展限制：隨著處理節點的總數增加，相關聯的網路額外負荷也會增加。每個涉及透過網路進行通訊的請求都可能增加延遲，並對平行化的好處產生負面影響。
- 回應時間瓶頸：對於要求在最終處理完成之前處理所有收件人的操作，整體系統的效能會受到最慢收件人的回應時間限制。
- 部分回應：當請求分散到多個收件人時，有些收件人可能會逾時。在這些情況下，實作應該向用戶端傳達回應不完整。您也可以使用 UI 前端顯示回應彙總詳細資訊。
- 資料一致性：當您跨多個收件人處理資料時，您必須仔細考慮資料同步和衝突解決技術，以確保最終彙總結果準確一致。

實作

高層級架構

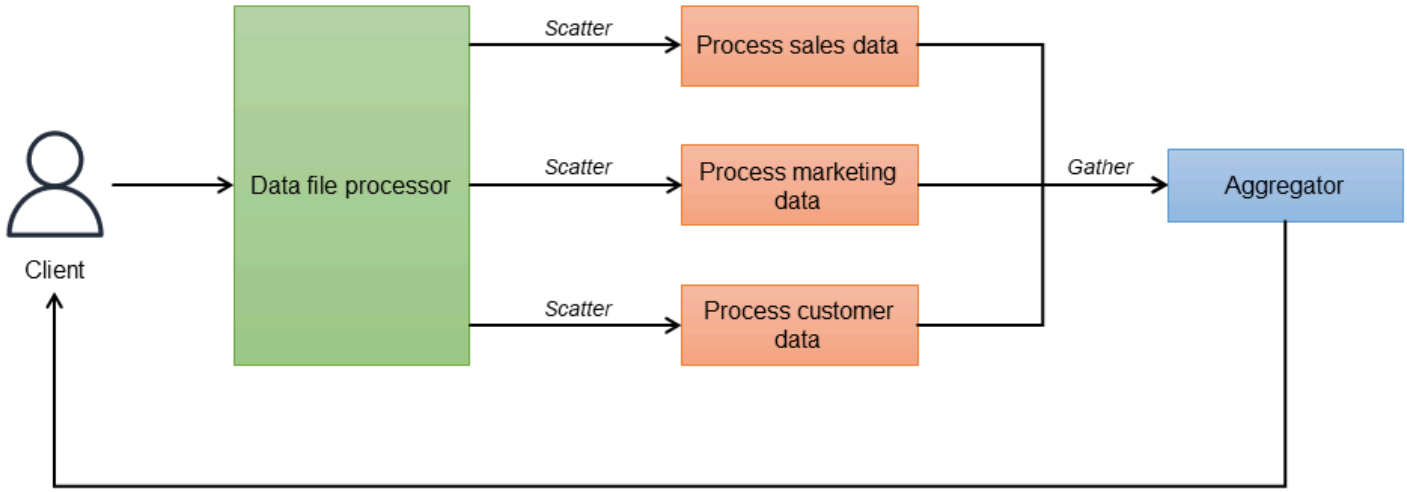
散佈集模式使用根控制器將請求分發給將處理請求的收件人。在散佈階段，此模式可以使用兩種機制來傳送訊息給收件人：

- 依分佈散佈：應用程式具有已知的收件人清單，必須呼叫才能取得結果。收件人可以是具有唯一函數的不同程序，或已向外擴展以分配處理負載的單一程序。如果任何處理節點逾時或顯示回應延遲，控制器可以將處理重新分配到另一個節點。
- 依競價散佈：應用程式會使用發佈訂閱模式，將訊息廣播給感興趣的收件人。在此情況下，收件人可以隨時訂閱訊息或退出訂閱。

依分佈散佈

在依分佈方法的散佈中，根控制器會將傳入的請求分割為獨立任務，並將其指派給可用的收件人 (散佈階段）。每個收件人（程序、容器或 Lambda 函數）在其運算上獨立並平行運作，並產生回應的一

部分。當收件人完成任務時，他們會將回應傳送到彙整工具 (收集階段）。彙總工具會結合部分回應，並將最終結果傳回給用戶端。下圖說明此工作流程。

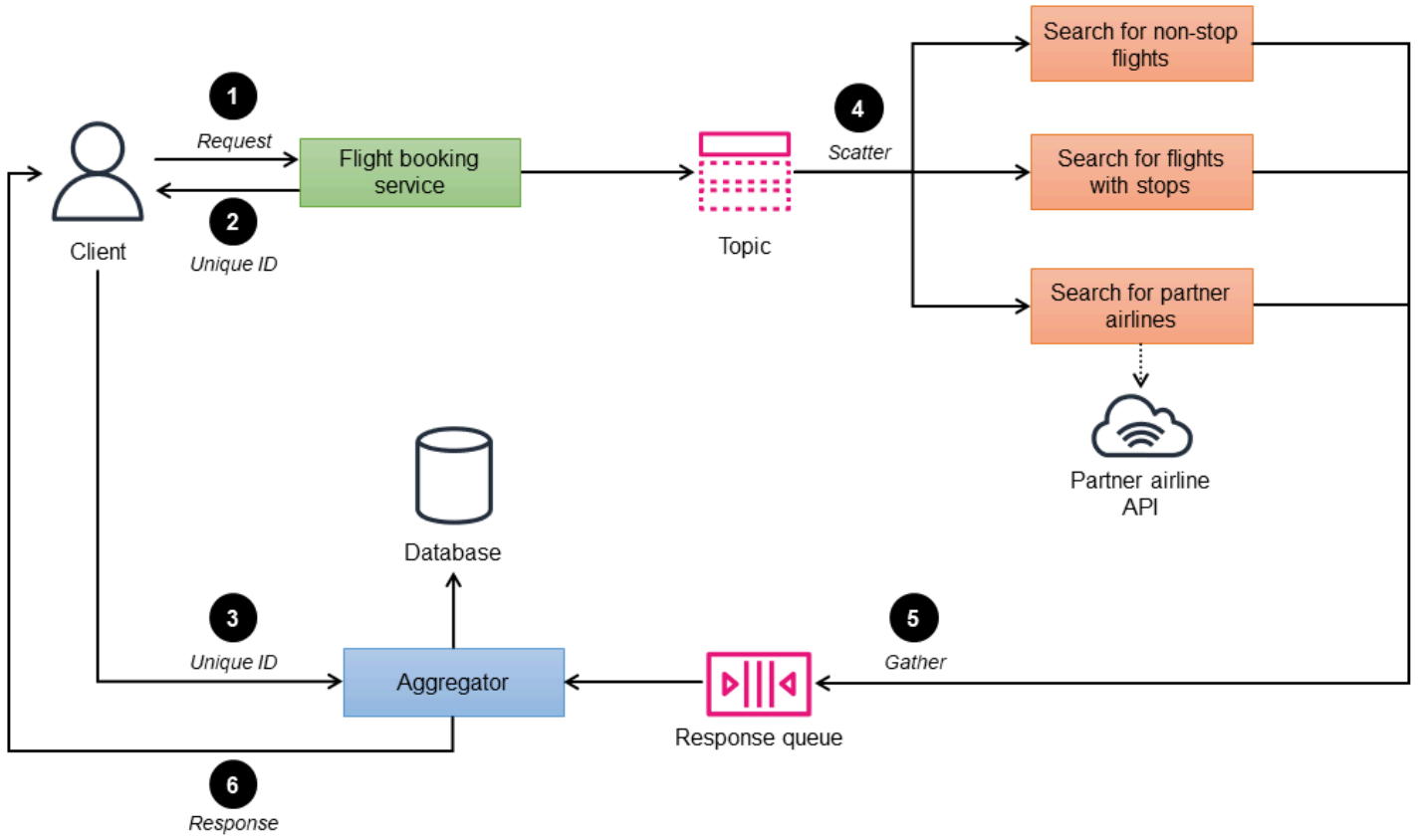


控制器（資料檔案處理器）會協調整組調用，並知道要呼叫的所有預訂端點。它可以設定逾時參數，以忽略花費太長的回應。傳送請求後，彙總工具會等待每個端點的回應傳回。若要實作彈性，每個微服務都可以部署多個執行個體以進行負載平衡。彙總工具會取得結果、將其合併為單一回應訊息，並在進一步處理之前移除重複的資料。系統會忽略逾時的回應。控制器也可以充當彙總工具，而不是使用單獨的彙總工具服務。

依競價散佈

如果控制器不知道收件人或收件人鬆散耦合，您可以透過競價方法使用散佈。在此方法中，收件人會訂閱主題，而控制器會將請求發佈至主題。收件人會將結果發佈到回應佇列。由於根控制器不知道收件人，因此收集程序會使用彙總工具（另一個傳訊模式）來收集回應，並將其分割成單一回應訊息。彙總工具會使用唯一的 ID 來識別一組請求。

例如，在下圖中，競價散佈法用於實作航空公司網站的航班預訂服務。該網站允許使用者搜尋和顯示來自航空公司及其合作夥伴航空公司的航班，並且必須即時顯示搜尋的狀態。航班預訂服務包含三種搜尋微服務：直飛航班、停靠航班，以及合作夥伴航空公司。合作夥伴航空公司搜尋會呼叫合作夥伴的 API 端點以取得回應。

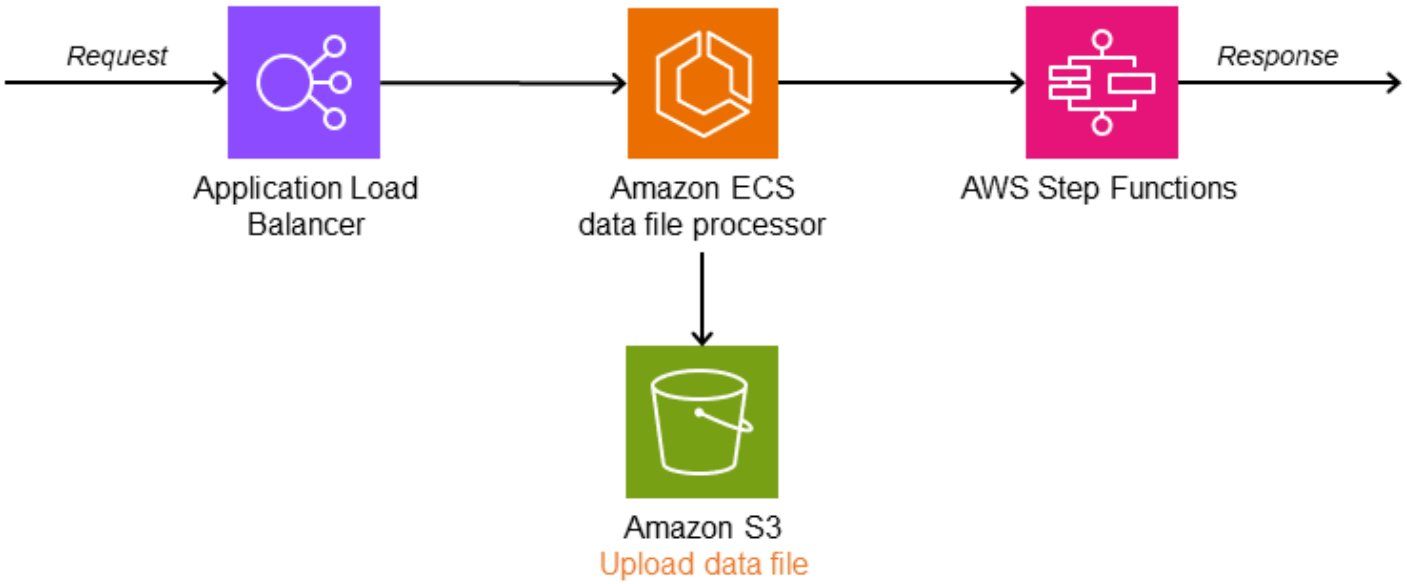


1. 航班預訂服務（控制器）接受搜尋條件做為用戶端的輸入，並處理請求並將其發佈至主題。
2. 控制器會使用唯一的 ID 來識別每個請求群組。
3. 用戶端會將唯一的 ID 傳送至步驟 6 的彙總工具。
4. 已訂閱預訂主題的預訂搜尋微服務會收到請求。
5. 微服務會處理請求，並將指定搜尋條件的座位可用性傳回給回應佇列。
6. 彙總工具會整理存放在暫時資料庫中的所有回應訊息、依唯一 ID 將航班分組、建立單一統一回應，並將其傳回給用戶端。

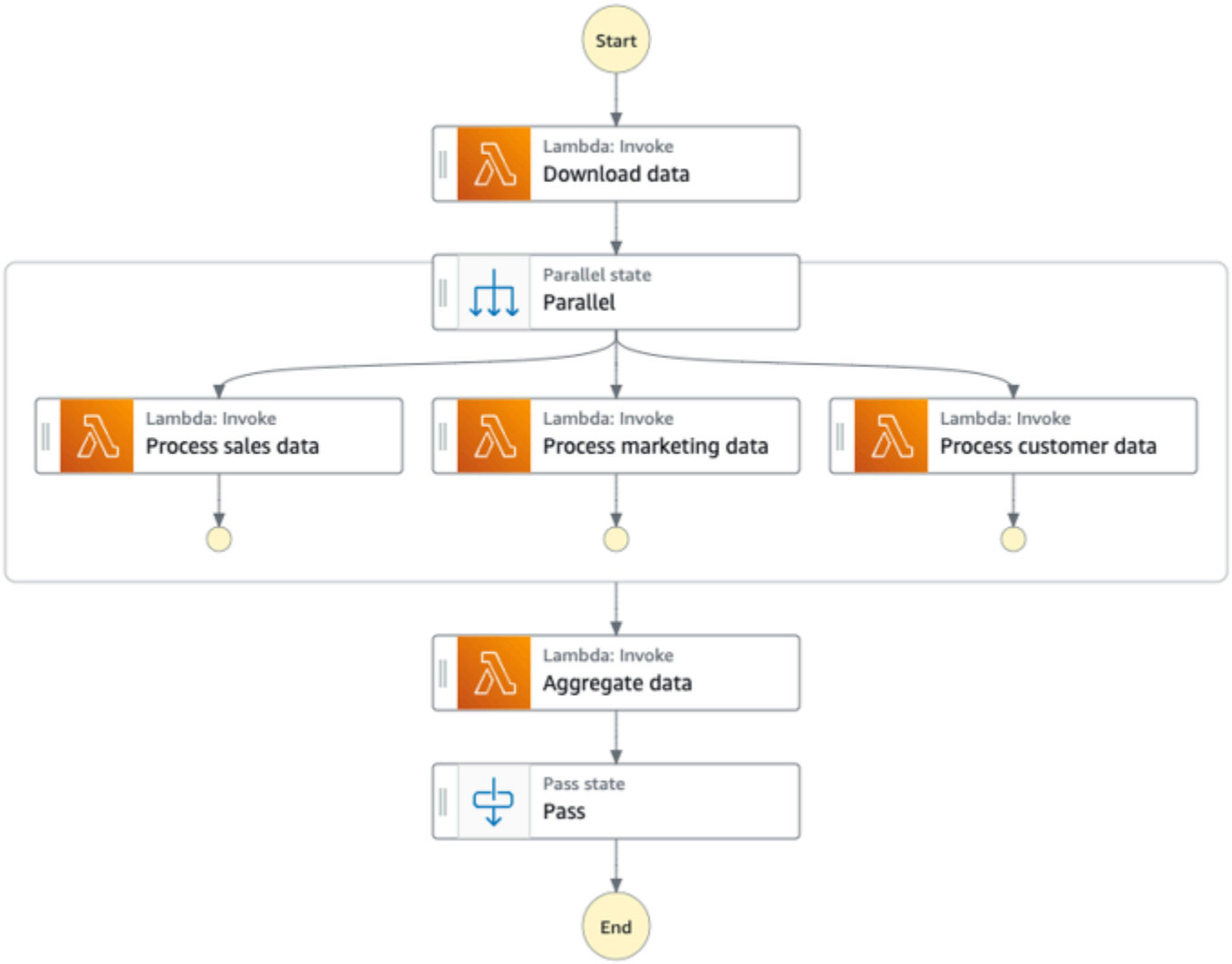
使用 實作 AWS 服務

依分佈散佈

在下列架構中，根控制器是資料檔案處理器 (Amazon ECS)，會將傳入的請求資料分割為個別 Amazon Simple Storage Service (Amazon S3) 儲存貯體，並啟動 AWS Step Functions 工作流程。工作流程會下載資料並啟動平行檔案處理。`Parallel` 狀態會等待所有任務傳回回應。 AWS Lambda 函數會彙總資料並將其儲存回 Amazon S3。

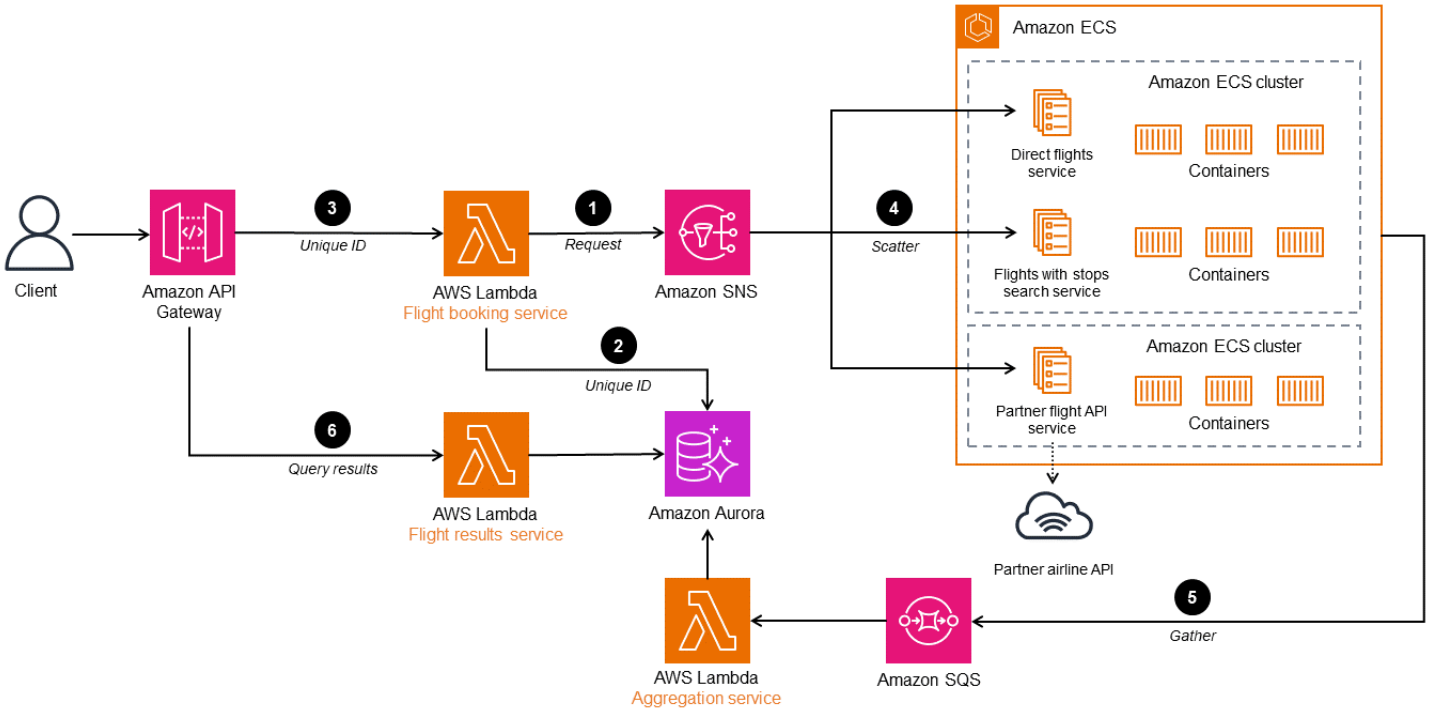


下圖說明具有 `Parallel` 狀態的 Step Functions 工作流程。



依競價散佈

下圖顯示依競價方法散佈的 AWS 架構。根控制器航班預訂服務會將航班搜尋請求分散至多個微服務。發佈訂閱頻道是使用 Amazon Simple Notification Service (Amazon SNS) 實作，Amazon Simple Notification Service 是用於通訊的受管傳訊服務。Amazon SNS 支援解耦微服務應用程式或直接與使用者通訊之間的訊息。您可以在 Amazon Elastic Kubernetes Service (Amazon EKS) 或 Amazon Elastic Container Service (Amazon ECS) 上部署收件人微服務，以獲得更好的管理和可擴展性。航班結果服務會將結果傳回給用戶端。它可以在 AWS Lambda 或其他容器協同運作服務中實作，例如 Amazon ECS 或 Amazon EKS。



1. 航班預訂服務（控制器）接受搜尋條件做為用戶端的輸入，並處理請求並將其發佈至 SNS 主題。
2. 控制器會將唯一 ID 發佈至 Amazon Aurora 資料庫，以識別請求。
3. 用戶端會將唯一 ID 傳送至步驟 6 的用戶端。
4. 已訂閱預訂主題的預訂搜尋微服務會收到請求。
5. 微服務會處理請求，並將指定搜尋條件的座位可用性傳回 Amazon Simple Queue Service (Amazon SQS) 中的回應佇列。彙總工具會整理所有回應訊息，並將其存放在暫時資料庫中。
6. 航班結果服務會依唯一 ID 將航班分組，建立單一統一的回應，並將其傳回給用戶端。

如果您想要將另一家航空公司搜尋新增至此架構，您可以新增訂閱 SNS 主題並發佈至 SQS 佇列的微服務。

總而言之，散佈收集模式可讓分散式系統達成高效率的平行處理、減少延遲，並順暢地處理非同步通訊。

GitHub 儲存庫

如需此模式範例架構的完整實作，請參閱 GitHub 儲存庫，網址為 https：//https://github.com/aws-samples/asynchronous-messaging-workshop/tree/master/code/lab-3。

研討會

- 解耦微服務研討會中的散佈集合實驗室

部落格參考

- 微服務的應用程式整合模式

相關內容

- 發佈訂閱模式

Strangler 無花果模式

意圖

Strangler fig 模式有助於逐步將單體應用程式遷移至微服務架構，並降低轉型風險和業務中斷。

動機

開發單體應用程式是為了在單一程序或容器中提供其大部分功能。程式碼會緊密耦合。因此，應用程式變更需要徹底重新測試，以避免迴歸問題。變更無法單獨測試，這會影響週期時間。隨著應用程式擁有更多功能，高複雜性可能會導致在維護上花費更多時間、增加上市時間，因此導致產品創新緩慢。

當應用程式擴展大小時，會增加團隊的認知負載，並可能導致團隊擁有權界限不明。無法根據負載擴展個別功能 - 必須擴展整個應用程式以支援尖峰負載。隨著系統老化，技術可能會變得過時，進而增加支援成本。單體舊版應用程式遵循開發時可用的最佳實務，且並非設計為可分發。

當單一應用程式遷移到微服務架構時，它可以分割成較小的元件。這些元件可以獨立擴展，可以獨立發行，也可以由個別團隊擁有。這會導致更高的變更速度，因為變更是本地化的，並且可以快速測試和發佈。變更的影響範圍較小，因為元件鬆散耦合且可個別部署。

透過重寫或重構程式碼，將整體完全取代為微服務應用程式，是一項巨大的任務，也是很大的風險。巨型遷移，其中整體在單一操作中遷移，帶來轉型風險和業務中斷。當應用程式正在重構時，新增新功能非常困難或甚至不可能。

解決此問題的一種方法是使用由 Martin Fowler 引進的 strangler fig 模式。此模式涉及逐步擷取功能，並在現有系統周圍建立新的應用程式，以移至微服務。整體中的功能會逐漸被微服務取代，應用程式使用者能夠逐步使用新遷移的功能。當所有功能移至新系統時，單體應用程式可以安全地停用。

適用性

在下列情況下使用 strangler fig 模式：

- 您想要將整體應用程式逐步遷移至微服務架構。
- 由於整體的大小和複雜性，大型巨型遷移方法具有風險。
- 企業想要新增新功能，而且無法等待轉換完成。
- 在轉換期間，最終使用者必須受到最小的影響。

問題和考量

- 程式碼基礎存取：若要實作 strangler fig 模式，您必須能夠存取整體應用程式的程式碼基礎。隨著功能從整體遷移，您需要進行次要程式碼變更，並在整體中實作反貪汙層，以將呼叫路由到新的微服務。您無法在沒有程式碼基礎存取的情況下攔截呼叫。程式碼基礎存取對於重新導向傳入請求也很重要，因為可能需要一些程式碼重構，以便代理層可以攔截遷移功能的呼叫，並將其路由到微服務。
- 不清楚的網域：系統過早分解可能成本高昂，特別是當網域不清楚時，而且可能使服務界限出錯時。網域驅動設計 (DDD) 是了解網域的機制，而事件風暴是判斷網域界限的技術。
- 識別微服務：您可以使用 DDD 作為識別微服務的關鍵工具。若要識別微服務，請尋找服務類別之間的自然分割。許多 服務將擁有自己的資料存取物件，並可輕鬆解耦。具有相關商業邏輯的服務和沒有或很少相依性的類別是微型服務的良好候選項目。您可以在分解整體之前重構程式碼，以防止緊密耦合。您也應該考慮合規要求、發行節奏、團隊的地理位置、擴展需求、使用案例驅動的技術需求，以及團隊的認知負載。
- 反貪汙層：在遷移過程中，當整體中的功能必須呼叫遷移為微服務的功能時，您應該實作反貪汙層 (ACL)，將每個呼叫路由至適當的微服務。為了分離和防止對整體中現有發起人進行變更，ACL 可作為轉接器或外觀，將呼叫轉換為較新的界面。本指南稍早的 ACL 模式實作一節會詳細說明。
- 代理層失敗：在遷移期間，代理層會攔截送至單體應用程式的請求，並將其路由至舊版系統或新系統。不過，此代理層可能會成為單一故障點或效能瓶頸。
- 應用程式複雜性：大型整體受益於 strangler fig 模式。對於完全重構的複雜性很低的小型應用程式，在微服務架構中重寫應用程式可能更有效率，而不是遷移應用程式。
- 服務互動：微服務可以同步或非同步通訊。需要同步通訊時，請考慮逾時是否會導致連線或執行緒集區耗用，進而導致應用程式效能問題。在這種情況下，請使用斷路器模式，針對可能會長時間失敗的操作傳回立即故障。您可以使用事件和訊息佇列來達成非同步通訊。
- 資料彙總：在微服務架構中，資料會分散到資料庫。需要資料彙總時，您可以在AWS AppSync前端使用 ，或在後端使用命令查詢責任隔離 (CQRS) 模式。
- 資料一致性：微型服務擁有其資料存放區，而單體應用程式也可能使用此資料。若要啟用共用，您可以使用佇列和代理程式，將新的微服務的資料存放區與單體應用程式的資料庫同步。不過，這可能會導致兩個資料存放區之間的資料備援和最終一致性，因此我們建議您將其視為策略解決方案，直到您可以建立資料湖等長期解決方案為止。

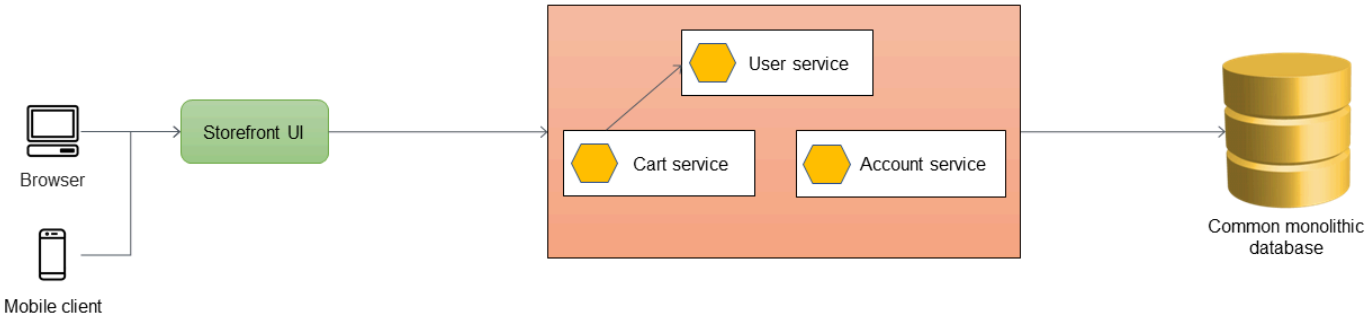
實作

在 strangler fig 模式中，您可以將特定功能取代為新的服務或應用程式，一次一個元件。代理層會攔截送至單體應用程式的請求，並將其路由至舊版系統或新系統。由於代理層會將使用者路由至正確的應用

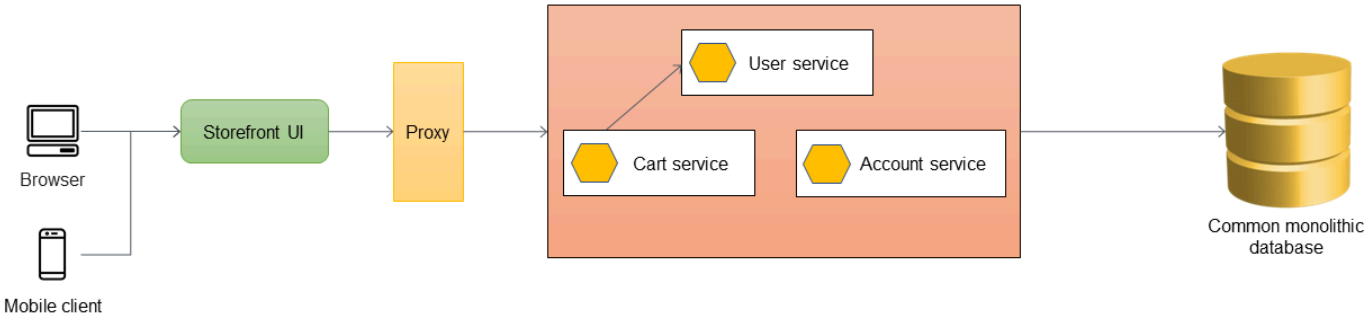
程式，因此您可以將功能新增至新系統，同時確保整體繼續運作。新系統最終會取代舊系統的所有功能，而且您可以停用它。

高層級架構

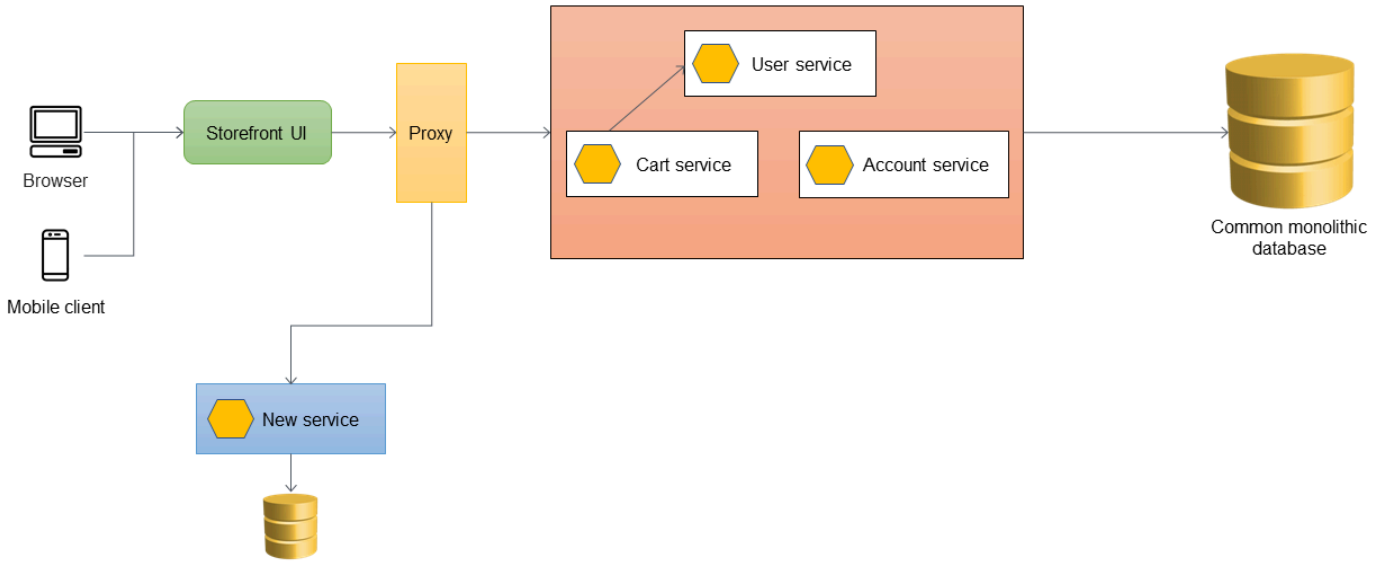
在下圖中，單一應用程式有三種服務：使用者服務、購物車服務和帳戶服務。購物車服務取決於使用者服務，而應用程式會使用單體關聯式資料庫。



第一步是在存放區 UI 和單體應用程式之間新增代理層。一開始，代理會將所有流量路由到整體應用程式。

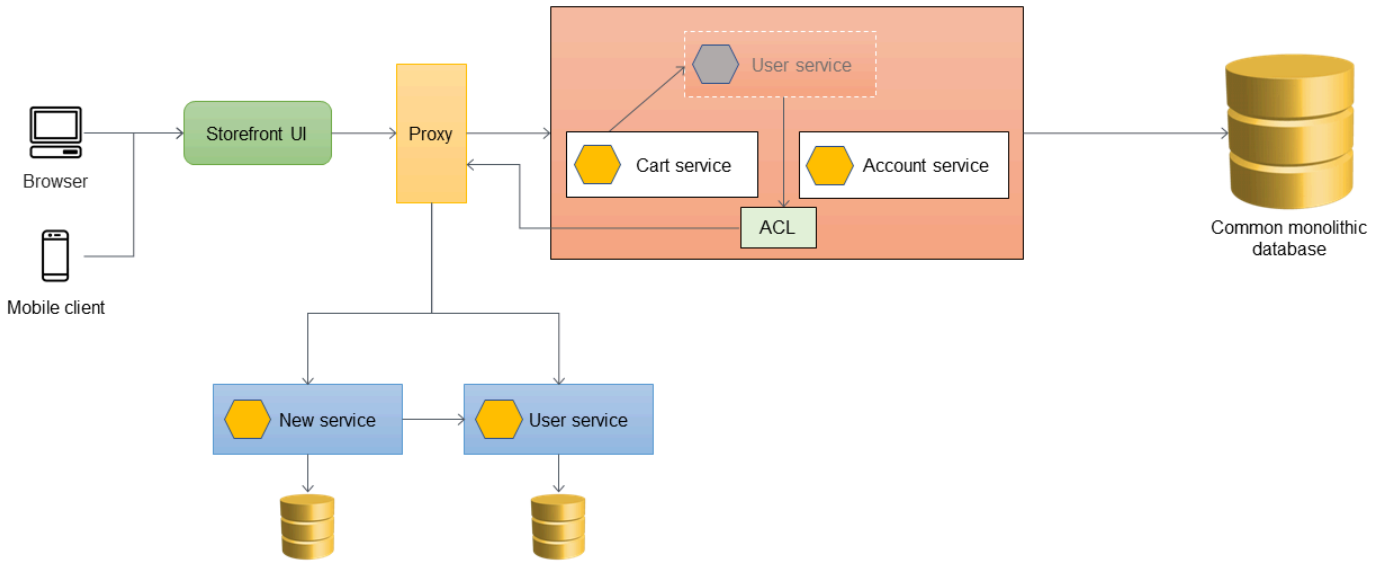


當您想要將新功能新增至應用程式時，您可以將它們實作為新的微服務，而不是將功能新增至現有的整體。不過，您可以繼續修正整體中的錯誤，以確保應用程式的穩定性。在下圖中，代理層會根據 API URL 將呼叫路由到整體或新的微服務。



新增反貪汙層

在下列架構中，使用者服務已遷移至微服務。購物車服務會呼叫使用者服務，但整體內不再提供實作。此外，新遷移服務的界面可能不符合其在單體應用程式內的先前界面。若要解決這些變更，請實作 ACL。在遷移過程中，當整體中的功能需要呼叫遷移為微服務的功能時，ACL 會將呼叫轉換為新界面，並將其路由至適當的微服務。

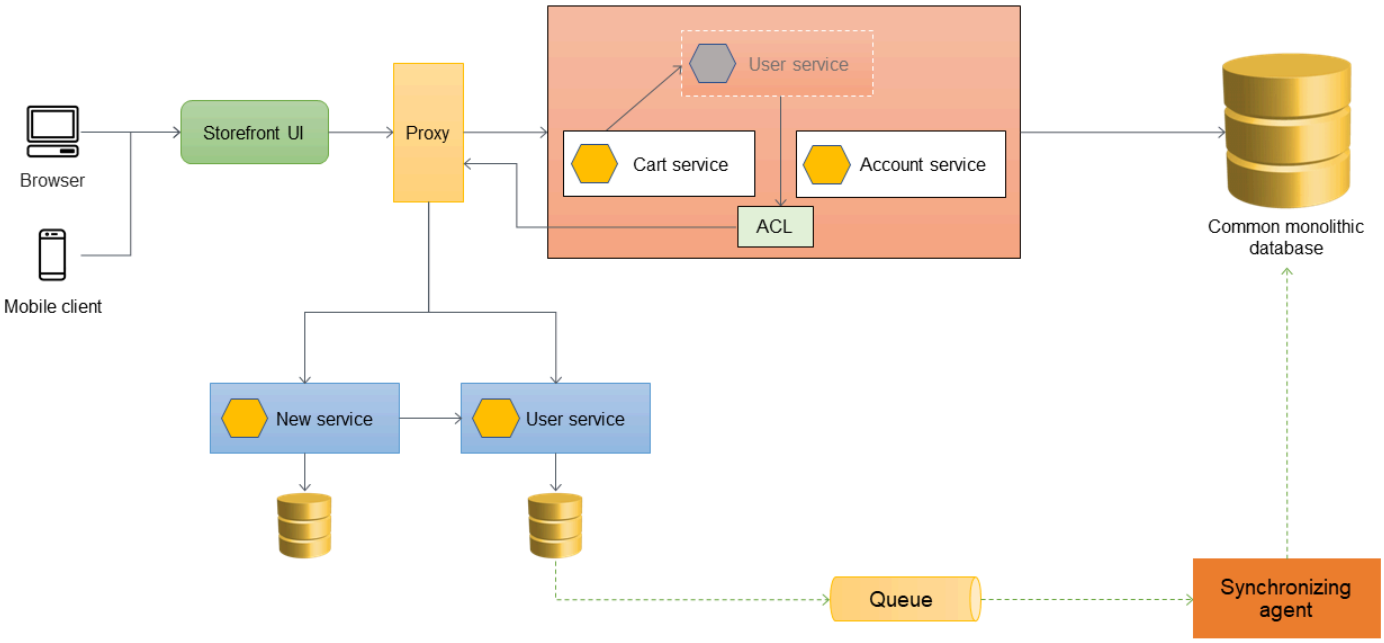


您可以在單體應用程式內實作 ACL，做為已遷移服務特有的類別；例如 `UserServiceFacade`或 `UserServiceAdapter`。在將所有相依服務遷移至微服務架構之後，必須停用 ACL。

當您使用 ACL 時，購物車服務仍會呼叫整體中的使用者服務，而使用者服務會透過 ACL 重新導向對微服務的呼叫。購物車服務仍應呼叫使用者服務，而不了解微服務遷移。需要這種鬆散耦合來減少迴歸和業務中斷。

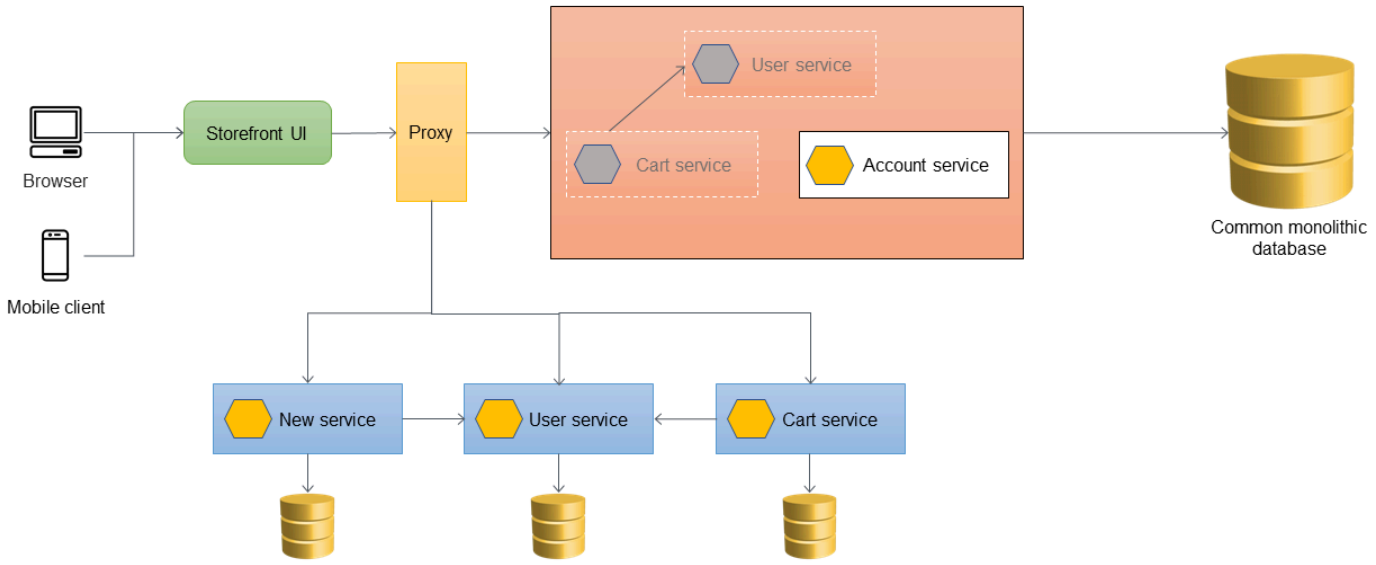
處理資料同步

最佳實務是，微型服務應該擁有其資料。使用者服務將其資料存放在自己的資料存放區中。它可能需要將資料與單體資料庫同步，以處理報告等相依性，並支援尚未準備好直接存取微服務的下游應用程式。單體應用程式也可能需要其他尚未遷移至微服務之函數和元件的資料。因此，新微服務和整體之間需要資料同步。若要同步資料，您可以在使用者微服務和單體資料庫之間引入同步代理程式，如下圖所示。使用者微服務會在資料庫更新時，將事件傳送至佇列。同步代理程式會接聽佇列，並持續更新整體資料庫。單體資料庫中的資料最終與正在同步的資料一致。

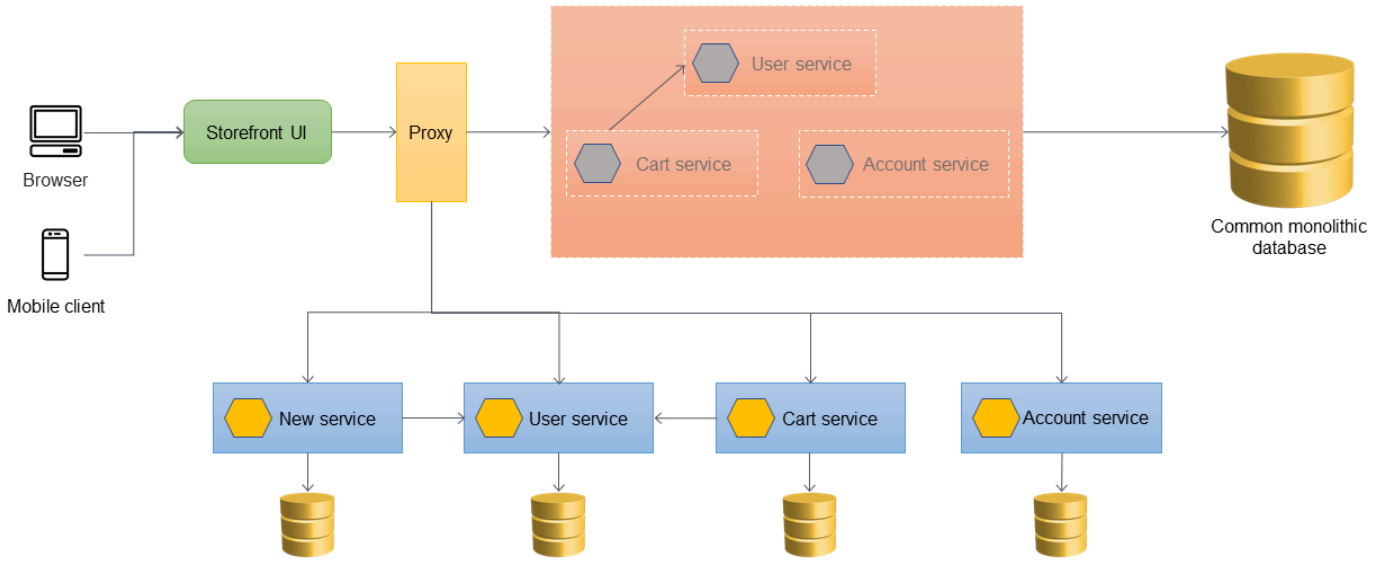


遷移其他服務

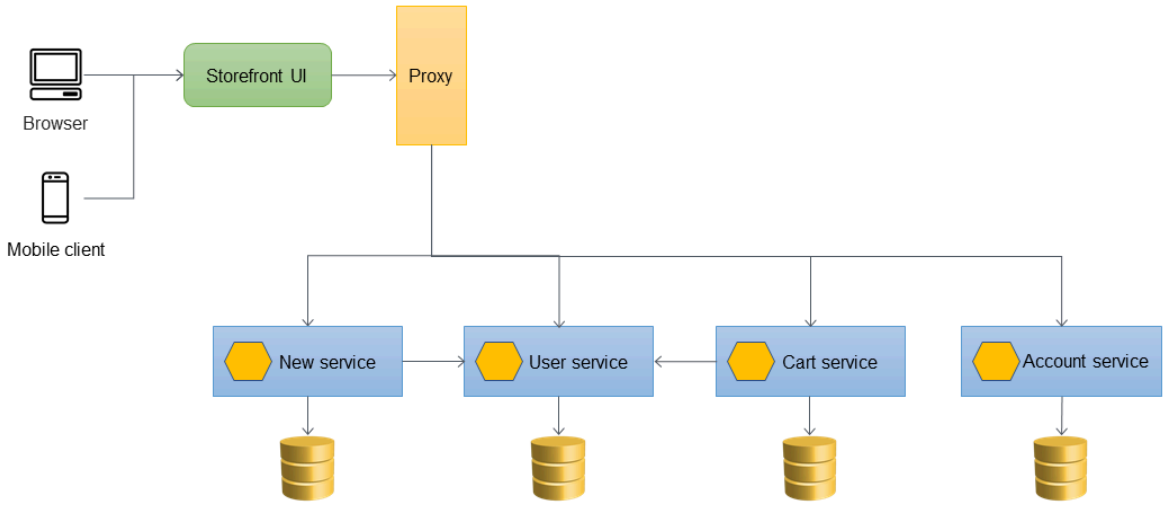
當購物車服務遷移出單體應用程式時，其程式碼會修改為直接呼叫新服務，因此 ACL 不會再路由這些呼叫。下圖說明此架構。



下圖顯示最終的扭曲狀態，其中所有 服務都已從整體遷移，且僅保留整體的骨架。歷史資料可以遷移到個別 服務擁有的資料存放區。可以移除 ACL，而且單體已準備好在此階段停用。



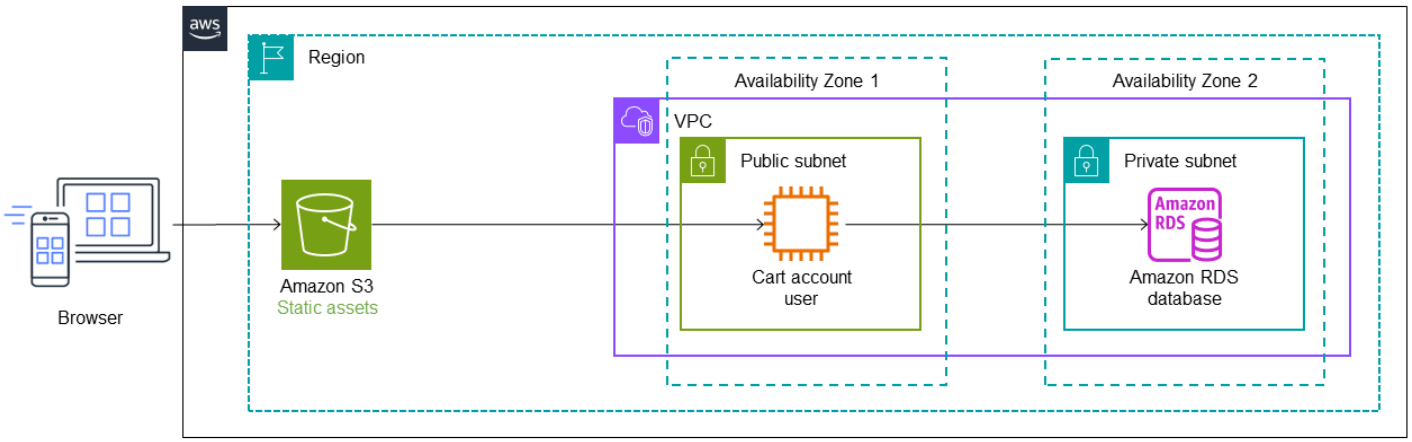
下圖顯示停用單體應用程式之後的最終架構。您可以根據您的應用程式需求，透過以資源為基礎的 URL（例如 `http://www.storefront.com/user`) 或透過自己的網域（例如 `http://user.storefront.com`) 託管個別微服務。如需使用主機名稱和路徑向上游消費者公開 HTTP APIs 的主要方法的詳細資訊，請參閱 API 路由模式一節。



使用 AWS 服務來實作

使用 API Gateway 做為應用程式代理

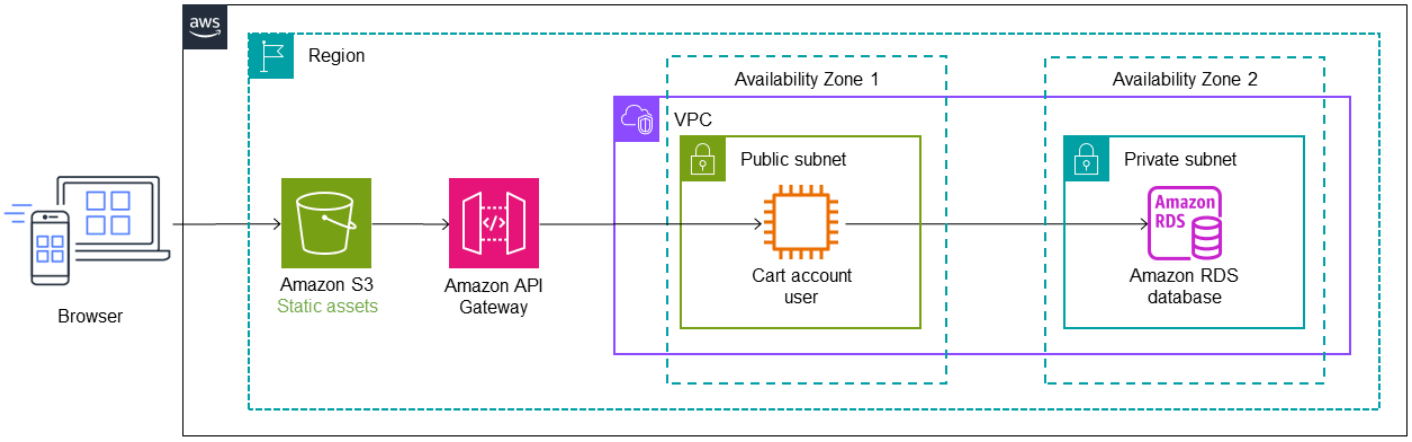
下圖顯示單體應用程式的初始狀態。假設它已 AWS 使用lift-and-shift策略遷移至 ，因此會在 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體上執行，並使用 Amazon Relational Database Service (Amazon RDS) 資料庫。為求簡化，架構使用單一虛擬私有雲端 (VPC) 搭配一個私有和一個公有子網路，讓我們假設微服務一開始會部署在相同的 中 AWS 帳戶。（生產環境中的最佳實務是使用多帳戶架構來確保部署獨立性。) EC2 執行個體位於公有子網路中的單一可用區域，RDS 執行個體則位於私有子網路中的單一可用區域。Amazon Simple Storage Service (Amazon S3) 存放靜態資產，例如網站的 JavaScript、CSS 和 React 檔案。



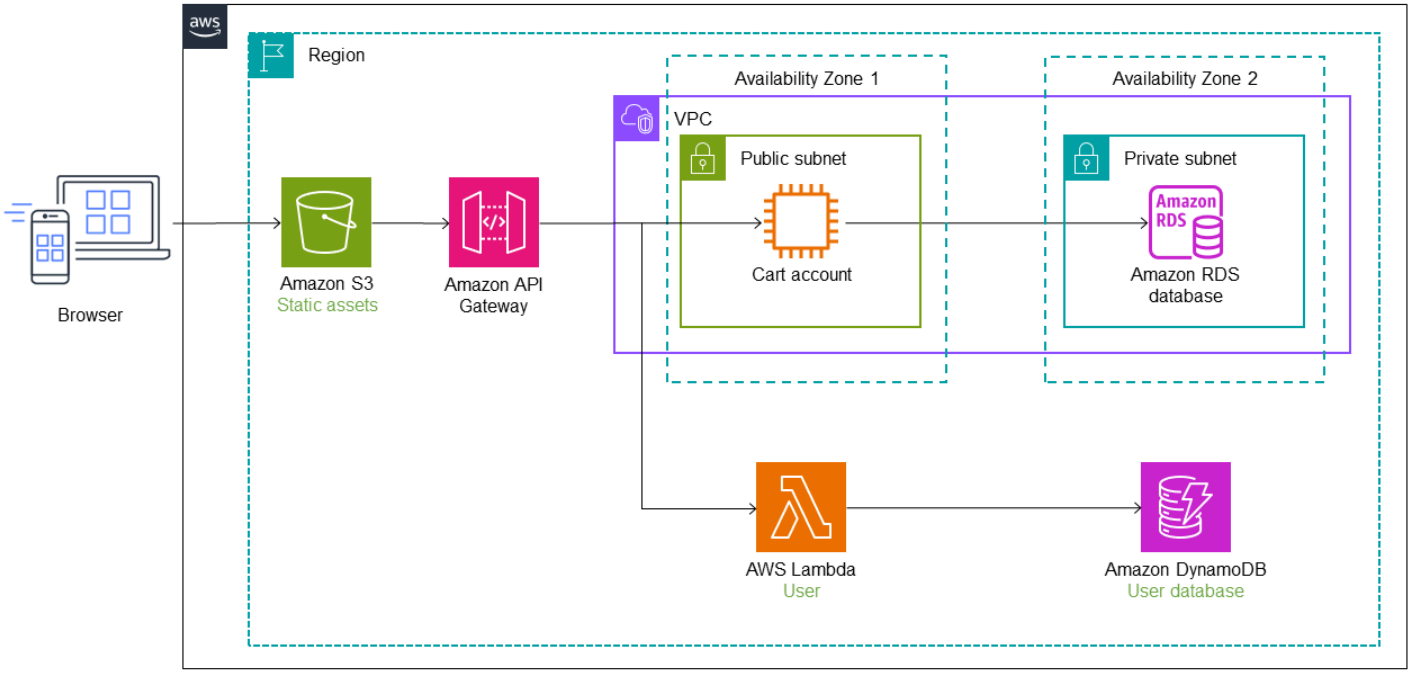
在下列架構中， AWS Migration Hub Refactor Spaces 會在單體應用程式前面部署 Amazon API Gateway。Refactor Spaces 會在您的帳戶內建立重構基礎設施，而 API Gateway 會做為將呼叫路由至

整體的代理層。一開始，所有呼叫都會透過代理層路由到單體應用程式。如前所述，代理層可能會成為單一故障點。不過，使用 API Gateway 做為代理可降低風險，因為它是無伺服器、多可用區域服務。

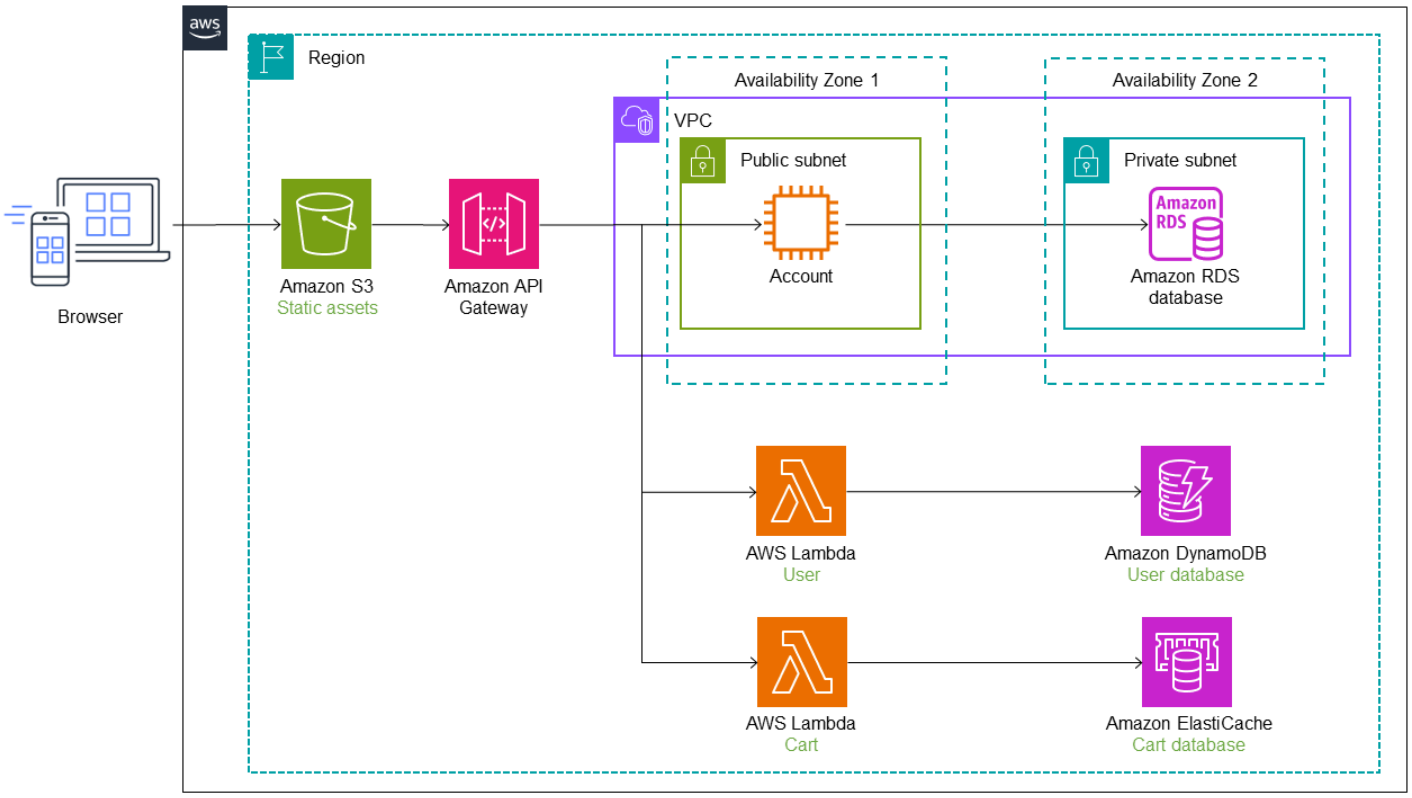
> **Note**
>
> AWS Migration Hub Refactor Spaces 自 2025 年 11 月 7 日起不再向新客戶開放。對於類似的功能 AWS Migration Hub Refactor Spaces，請探索 AWS Transform。



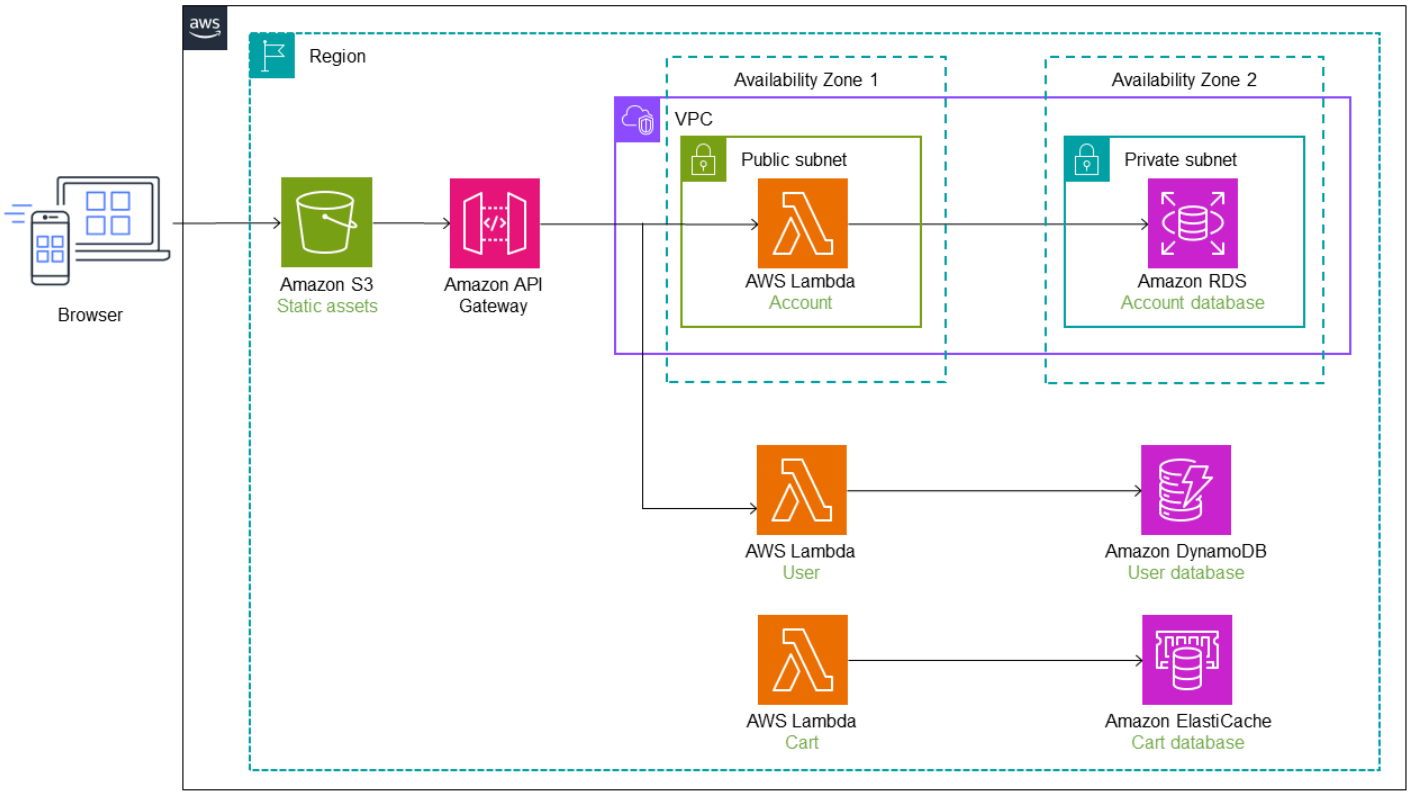
使用者服務會遷移至 Lambda 函數，Amazon DynamoDB 資料庫會存放其資料。Lambda 服務端點和預設路由會新增至 Refactor Spaces，而 API Gateway 會自動設定為將呼叫路由到 Lambda 函數。



在下圖中，購物車服務也已從整體遷移到 Lambda 函數中。額外的路由和服務端點會新增至重構空間，而流量會自動切換到 `Cart` Lambda 函數。Lambda 函數的資料存放區由 Amazon ElastiCache 管理。單體應用程式仍會與 Amazon RDS 資料庫一起保留在 EC2 執行個體中。



在下圖中，最後一個服務（帳戶）會從整體遷移到 Lambda 函數。它會繼續使用原始 Amazon RDS 資料庫。新的架構現在有三個具有不同資料庫的微服務。每個服務使用不同類型的資料庫。這種使用專用資料庫來滿足微服務特定需求的概念稱為多槽持久性。Lambda 函數也可以以不同的程式設計語言實作，具體取決於使用案例。在重構期間，重構空間會將流量的切換和路由自動化到 Lambda。這可節省建置器建構、部署和設定路由基礎設施所需的時間。

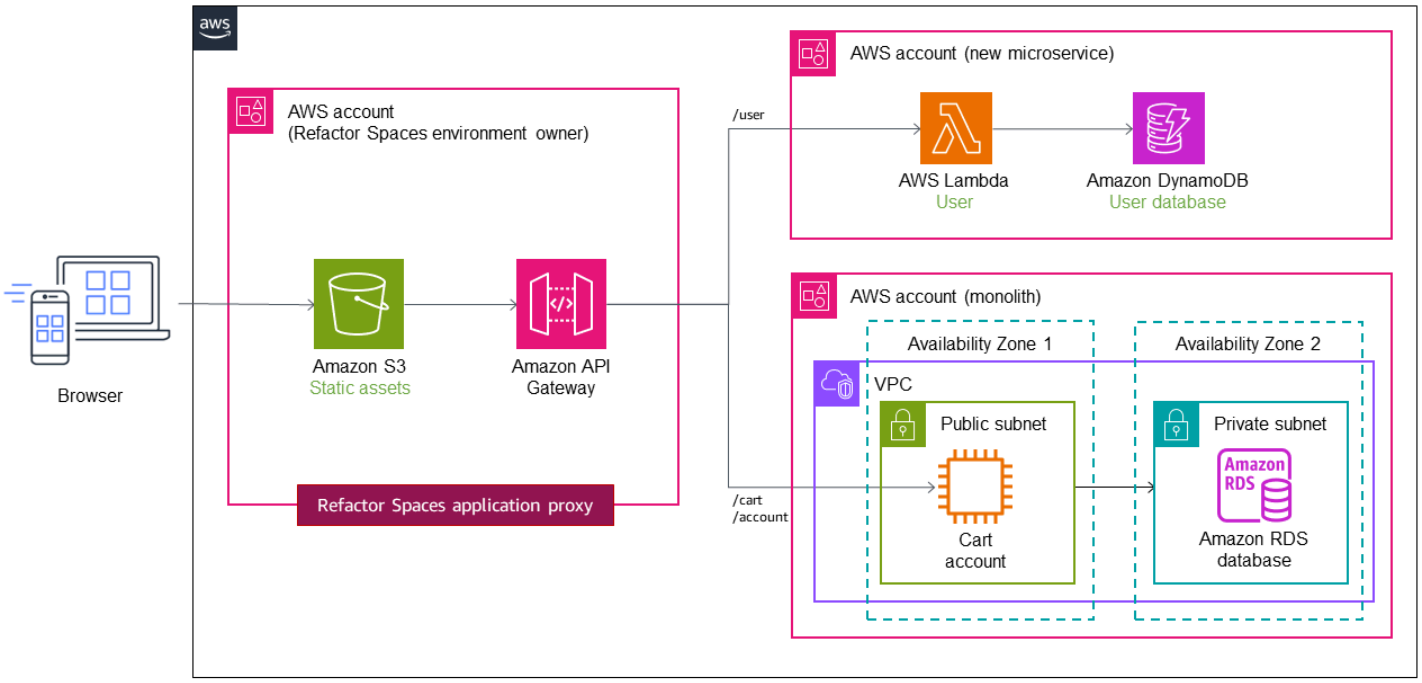


使用多個帳戶

在先前的實作中，我們為單體應用程式使用具有私有和公有子網路的單一 VPC，為了簡化 AWS 帳戶起見，我們在相同的 中部署微服務。不過，在實際案例中很少發生這種情況，其中微服務通常部署在多個 中 AWS 帳戶 ，以實現部署獨立性。在多帳戶結構中，您需要設定將流量從整體路由到不同帳戶中的新服務。

Refactor Spaces 可協助您建立和設定 AWS 基礎設施，以從單體應用程式路由 API 呼叫。Refactor Spaces 會在您的帳戶 AWS 內協調 API Gateway、Network Load Balancer 和資源型 AWS Identity and Access Management (IAM) 政策，做為其應用程式資源的一部分。您可以透明地將單一帳戶 AWS 帳戶 或跨多個帳戶的新服務新增至外部 HTTP 端點。所有這些資源都會在您的 中協調 AWS 帳戶 ，並在部署後進行自訂和設定。

假設使用者和購物車服務部署到兩個不同的帳戶，如下圖所示。當您使用 Refactor Spaces 時，只需要設定服務端點和路由。Refactor Spaces 會將 API Gateway–Lambda 整合和 Lambda 資源政策的建立自動化，因此您可以專注於安全地重構整體的服務。



如需使用 Refactor Spaces 的影片教學課程，請參閱使用 遞增重構應用程式 AWS Migration Hub Refactor Spaces。

研討會

- 迭代應用程式現代化研討會

部落格參考

- AWS Migration Hub Refactor Spaces
- 在 上深入探索 AWS Migration Hub Refactor Spaces
- 部署管道參考架構和參考實作

相關內容

- API 路由模式
- 重構 Spaces 文件

交易寄件匣模式

意圖

交易式寄件匣模式可解決當單一作業涉及資料庫寫入作業和訊息或事件通知時，分散式系統中所發生的雙重寫入作業問題。當應用程式寫入兩個不同的系統時，就會發生雙重寫入作業；例如，當微服務需要保留資料庫中的資料並傳送訊息以通知其他系統時。其中一項操作失敗可能會導致資料不一致。

動機

當微服務在資料庫更新之後傳送事件通知時，這兩項作業應以原子方式執行，以確保資料一致性和可靠性。

- 如果資料庫更新成功，但事件通知失敗，下游服務將無法察覺變更，而且系統可能會進入不一致的狀態。
- 如果資料庫更新失敗，但事件通知已傳送，資料可能會損毀，這可能會影響系統的可靠性。

適用性

出現下列情況時，請使用交易式寄件匣模式：

- 您正在構建一個事件驅動的應用程式，其中資料庫更新會啟動事件通知。
- 您想要確保涉及兩項服務的作業中的原子性。
- 您想要導入事件來源模式。

問題和考量

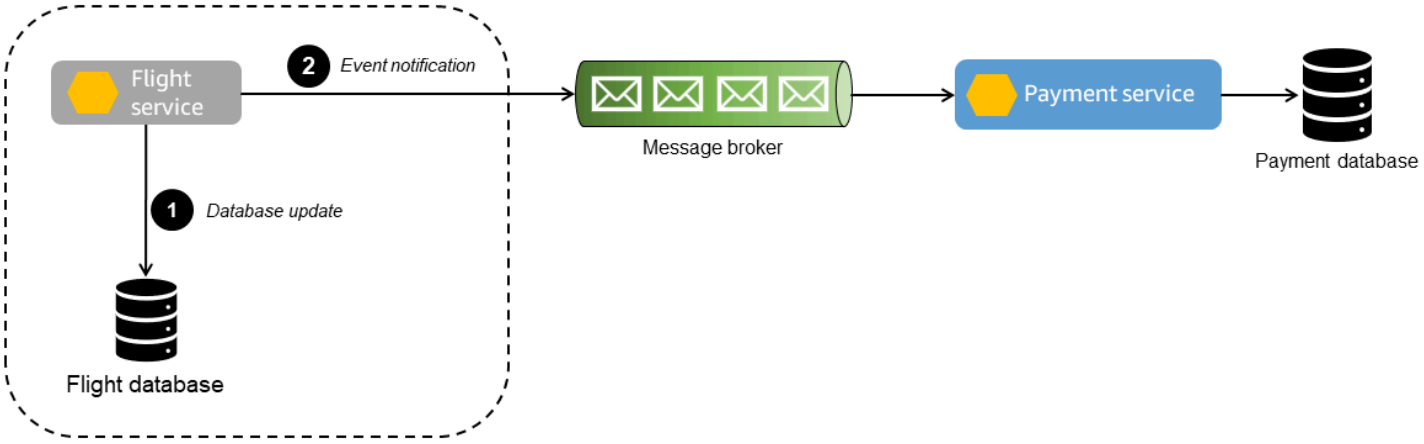
- 重複訊息：事件處理服務可能會傳送重複的訊息或事件，因此建議您追蹤已處理的訊息，讓消費服務成為等冪性。
- 通知順序：以服務更新資料庫的相同順序傳送訊息或事件。這對於事件來源模式而言相當重要，此模式是您可以在其中使用事件存放區來進行資料存放區的時間點復原。如果順序不正確，可能會影響資料的品質。如果未保留通知順序，則最終一致性和資料庫回復可以解決此問題。
- 交易回復：如果交易遭回復，請勿傳送事件通知。

- 服務層級交易處理：如果交易橫跨需要資料存放區更新的服務，請使用 系列事件協同運作模式來維持整個資料存放區的資料完整性。

實作

高層級架構

下列序列圖會顯示雙重寫入作業期間發生的事件順序。



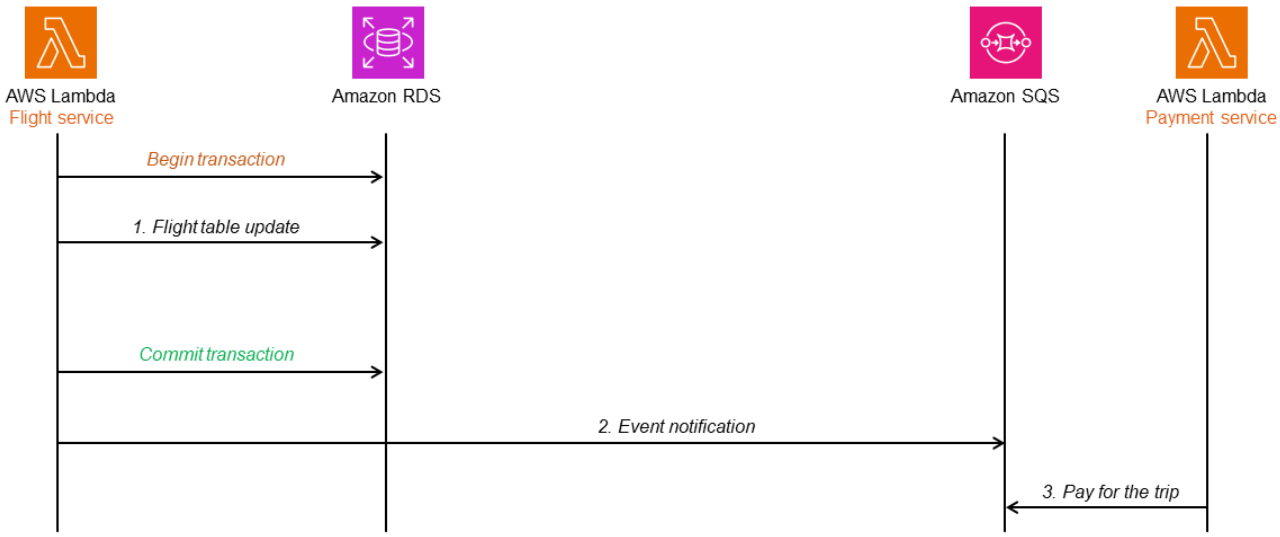
1. 飛行服務會寫入資料庫，並向付款服務傳送事件通知。
2. 訊息代理程式會將訊息和事件傳送至付款服務。訊息代理程式中的任何失敗都會導致付款服務無法接收更新。

如果飛行資料庫更新失敗但已傳送通知，則付款服務會根據事件通知處理付款。這將導致下游資料不一致。

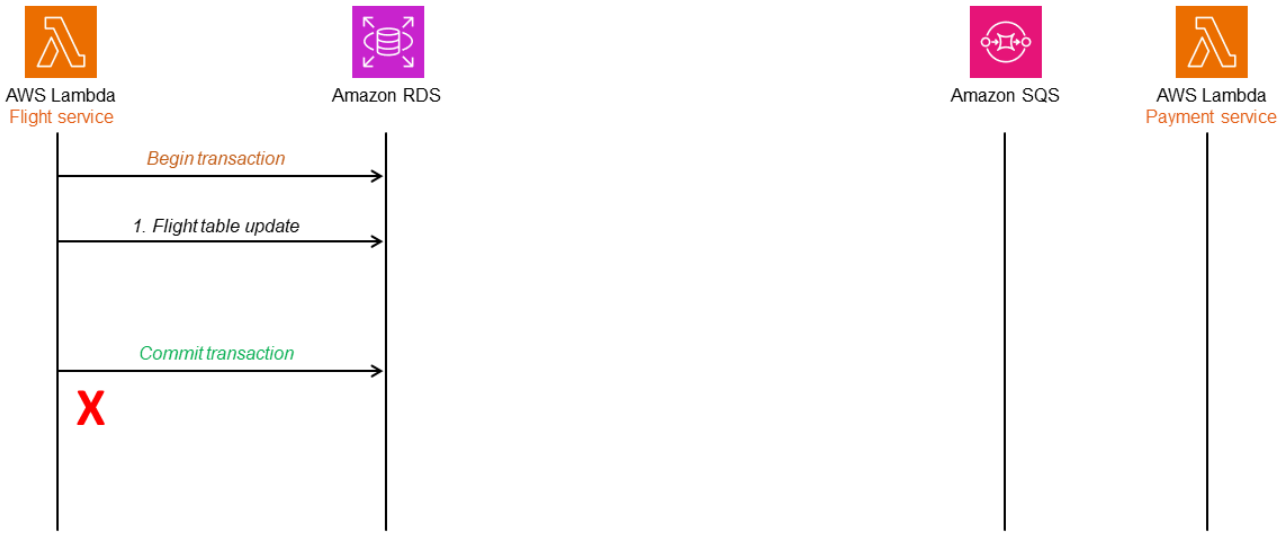
使用 AWS 服務來實作

為了示範序列圖中的模式，我們將使用下列 AWS 服務，如下圖所示。

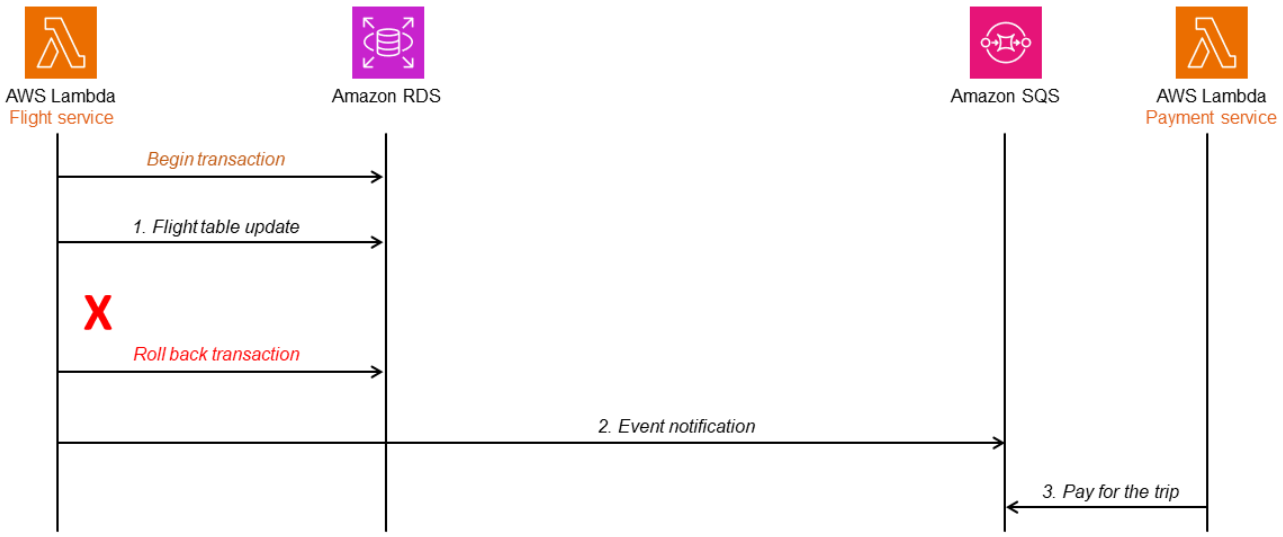
- 微服務是透過使用 AWS Lambda 來實作。
- 主要資料庫是由 Amazon Relational Database Service (Amazon RDS) 所管理。
- Amazon Simple Queue Service (Amazon SQS) 充當接收事件通知的訊息代理程式。



如果飛行服務在遞交該筆交易後失敗，此情況可能會導致事件通知無法傳送。



不過，交易可能會失敗並回復，但可能仍會傳送事件通知，導致付款服務處理付款。



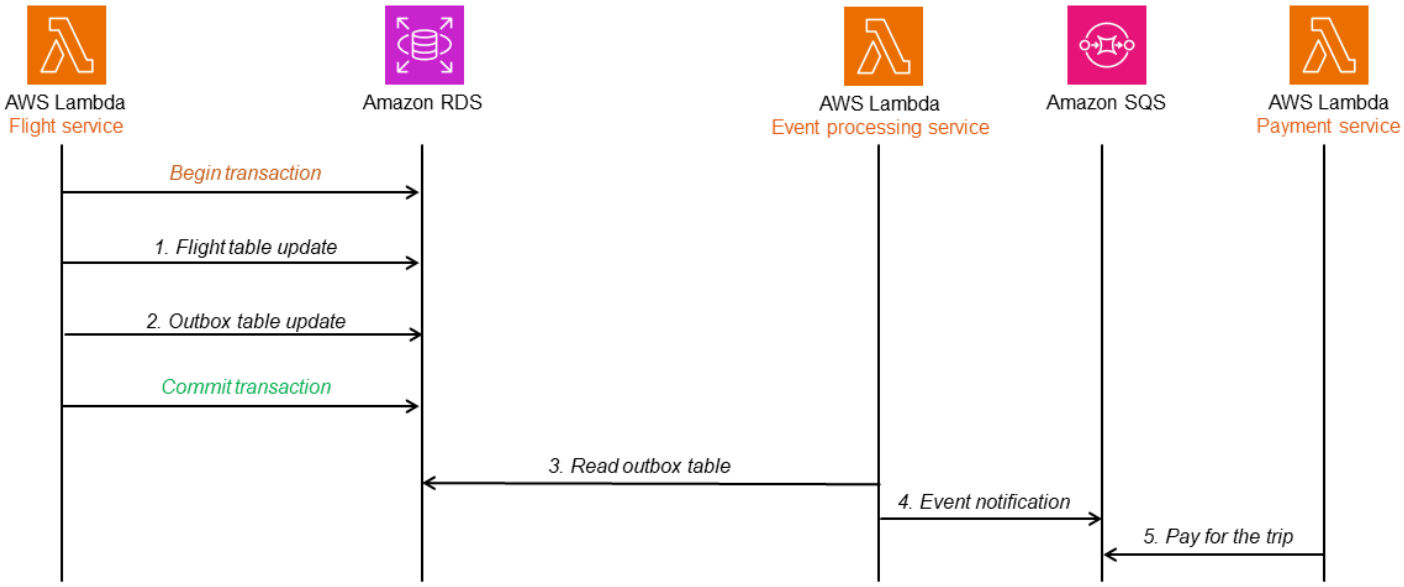
若要解決這個問題，您可以使用寄件匣資料表或變更資料擷取 (CDC)。下列區段將探討這兩個選項，以及您可以如何使用 AWS 服務來實作這些選項。

將寄件匣資料表與關聯式資料庫搭配使用

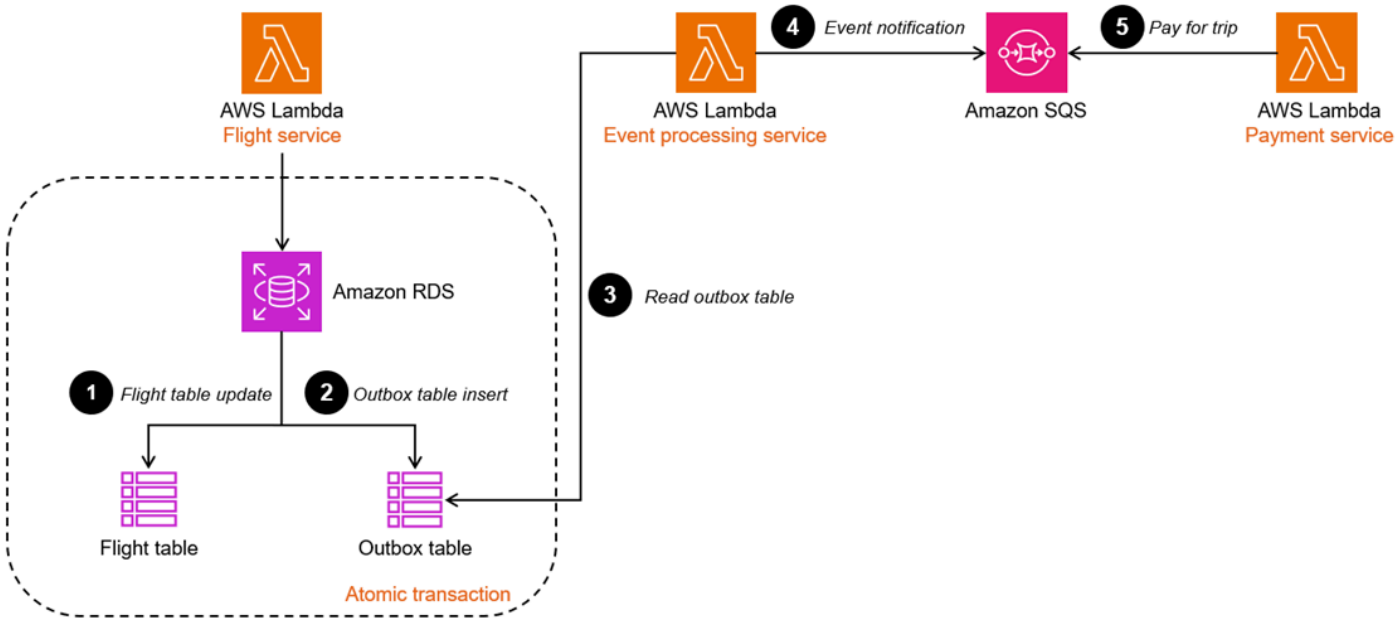
寄件匣資料表會儲存來自飛行服務的所有事件與時間戳記和序號。

當飛行資料表更新時，同一筆交易中的寄件匣資料表也會隨之更新。另一個服務 (例如，事件處理服務) 會從寄件匣資料表讀取，然後將事件傳送至 Amazon SQS。Amazon SQS 會將有關事件的訊息傳送至付款服務，以便進一步處理。Amazon SQS 標準佇列可保證訊息至少交付一次，而且不會遺失。但是，當您使用 Amazon SQS 標準佇列時，相同的訊息或事件可能會傳遞多次，因此您應確保事件通知服務是等冪的 (也就是說，多次處理相同訊息不會產生不良影響)。如果您需要只處理訊息一次，並使用訊息排序，則可以先輸入、先出 (FIFO) 佇列來使用 Amazon SQS。

如果飛行資料表更新失敗或寄件匣資料表更新失敗，則整筆交易都會回復，因此不會有不一致的下游資料。



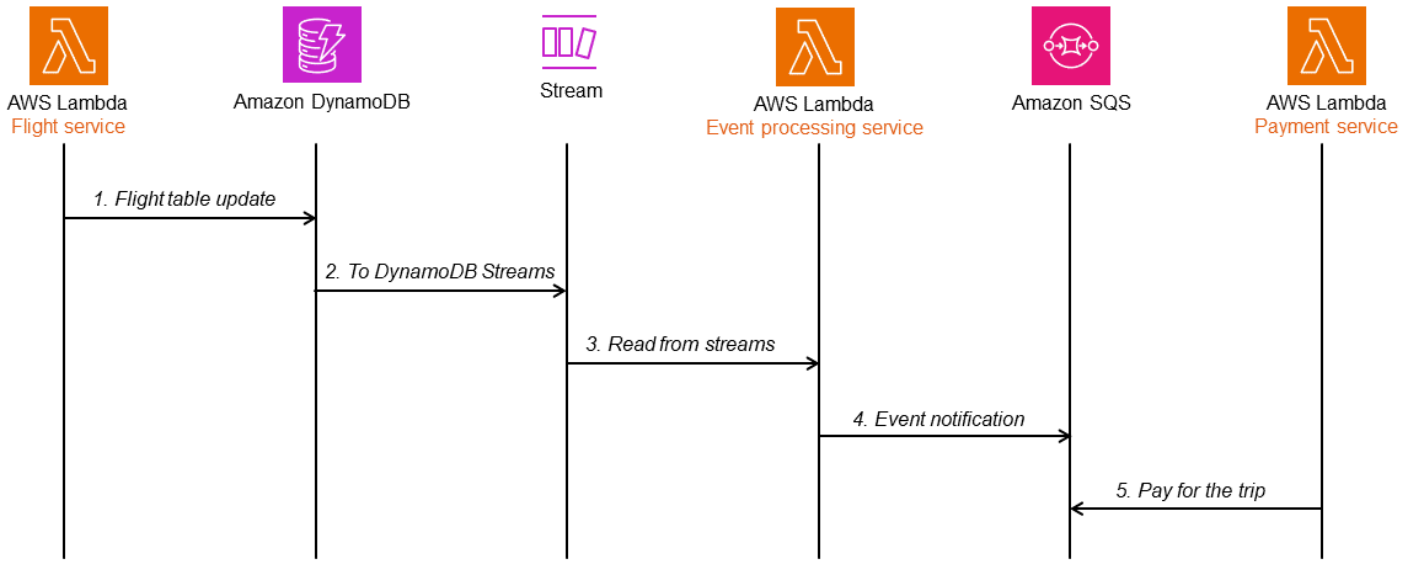
在下列圖表中，交易寄件匣架構是使用 Amazon RDS 資料庫來實作。當事件處理服務讀取寄件匣資料表時，它只會辨識屬於已遞交 (成功) 交易一部分的資料列，然後將事件的訊息置於 SQS 佇列中，付款服務會讀取此佇列以供進一步處理。此設計可解決雙重寫入作業問題，並使用時間戳記和序號來保留訊息和事件的順序。



使用變更資料擷取 (CDC)

某些資料庫支援發布項目層級修改，以擷取變更的資料。您可以識別已變更的項目，並據以傳送事件通知。這樣可以節省建立另一個資料表來追蹤更新的開銷。由飛行服務發起的事件會儲存在相同項目的另一個屬性中。

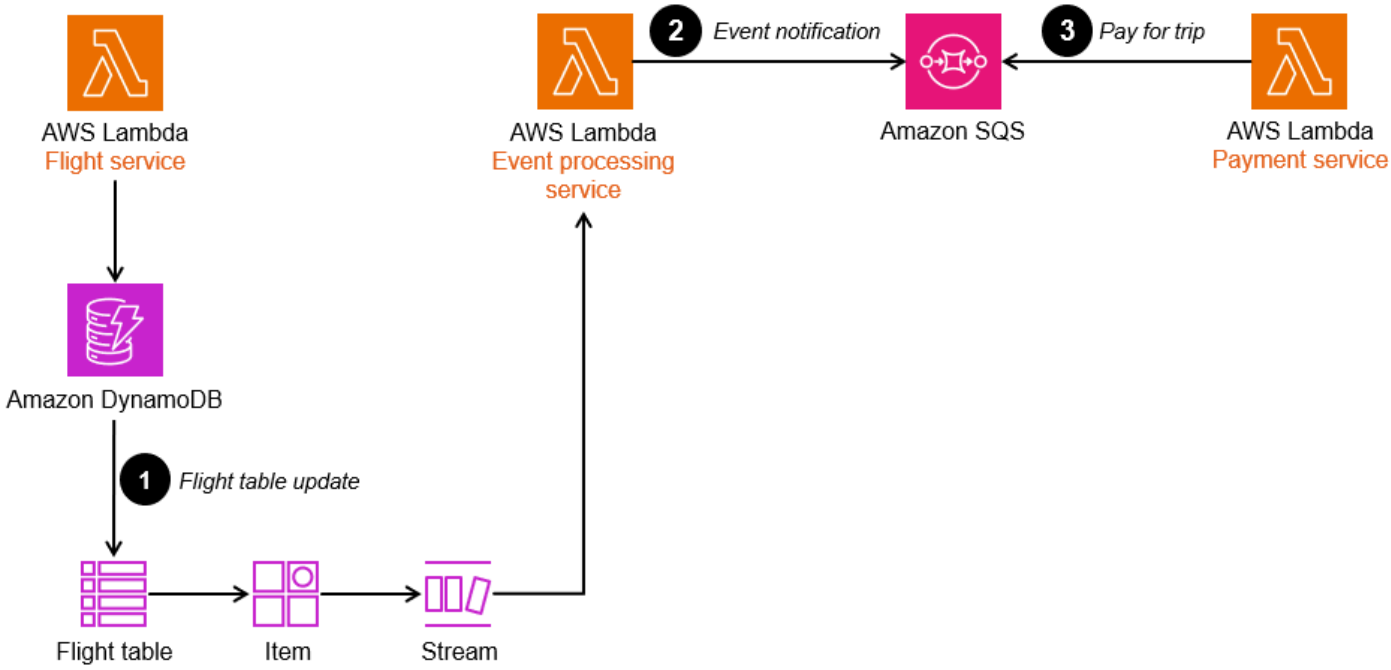
Amazon DynamoDB 是一個支援 CDC 更新的鍵值 NoSQL 資料庫。在下列順序圖中，DynamoDB 會將項目層級修改發布至 Amazon DynamoDB Streams。事件處理服務會從串流讀取，並將事件通知發布至付款服務，以便進一步處理。



DynamoDB Streams 流會使用時間排序序列擷取 DynamoDB 表中與項目層級變更相關的資訊流程。

您可以在 DynamoDB 資料表上啟用串流，藉此實作交易寄件匣模式。事件處理服務的 Lambda 函數與這些串流相關聯。

- 更新航班表時，DynamoDB Streams 會擷取變更的資料，而事件處理服務會輪詢串流以取得新記錄。
- 當新的串流記錄可供使用時，Lambda 函數會同步將事件的訊息放在 SQS 佇列中，以供進一步處理。您可以視需要將屬性新增至 DynamoDB 項目，以擷取時間戳記和序列號，以改善實作的穩健性。



範本程式碼

使用寄件匣資料表

本節中的範例程式碼說明如何使用寄件匣資料表來實作交易寄件匣模式。若要檢視完整的程式碼，請參閱 GitHub 儲存庫以取得此範例。

下面的代碼片段將 `Flight` 實體和 `Flight` 事件保存在單一交易中的各自資料表內的資料庫中。

```
@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;
    }
```

另一項服務負責定期掃描寄件匣資料表中是否有新事件、將事件傳送至 Amazon SQS，並在 Amazon SQS 成功回應時將其從資料表中刪除。輪詢速率可在 `application.properties` 檔案中設定。

```
@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);
        }
    }
```

使用變更資料擷取 (CDC)

本節中的範例程式碼說明如何使用 DynamoDB 的變更資料擷取 (CDC) 功能來實作交易寄件匣模式。若要檢視完整的程式碼，請參閱 GitHub 儲存庫以取得此範例。

下列 AWS Cloud Development Kit (AWS CDK) 程式碼片段會建立 DynamoDB 航班資料表和 Amazon Kinesis 資料串流 (`cdcStream`)，並設定航班資料表將其所有更新傳送至串流。

```
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,
    }

});
```

下列程式碼片段和組態定義了彈簧雲端串流函數，可擷取 Kinesis 串流中的更新，並將這些事件轉送至 SQS 佇列以進行進一步處理。

```
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 儲存庫

如需此模式範例架構的完整實作，請參閱 GitHub 存放庫，網址為 https://github.com/aws-samples/transactional-outbox-pattern。

資源

參考

- AWS 架構中心
- AWS 開發人員中心
- Amazon Builders Library

工具

- AWS Well-Architected Tool
- AWS App2Container
- AWS 適用於 .NET 的微服務擷取器

方法

- 十二要素應用程式 (ePub by Adam Wiggins)
- Nygard、Michael T. Release It！：設計和部署生產就緒軟體。第 2 版。Raleigh， NC：Pragmatic Bookshelf， 2018。
- Polyglot 持續性 (Martin Fowler 的部落格文章）
- StranglerFigApplication （作者為 Martin Fowler 的部落格文章）

文件歷史紀錄

下表描述了本指南的重大變更。如果您想收到有關未來更新的通知，可以訂閱 RSS 摘要。

變更	描述	日期
全新模式	新增兩種新模式：六邊形架構和散佈集合。	2024 年 5 月 7 日
新增程式碼範例	將變更資料擷取 (CDC) 使用案例的範例程式碼新增至交易寄件匣模式。	2024 年 2 月 23 日
新增程式碼範例	• 以範例程式碼更新了交易寄件匣模式。 • 移除了協同運作與編排模式區段，這些模式由系列事件編排和系列事件協同運作取代。	2023 年 11 月 16 日
全新模式	加入了三種新模式：系列事件編排、發布-訂閱，以及事件來源。	2023 年 11 月 14 日
更新	更新了 Strangler Fig 模式模式實作區段。	2023 年 10 月 2 日
初次出版	第一個發行版本包含八種設計模式：防損毀層 (ACL)、API 路由、斷路器、協調與編排、使用輪詢重試、系列事件協同運作、Strangler Fig 和交易寄件匣。	2023 年 7 月 28 日

AWS 規範性指引詞彙表

以下是 AWS Prescriptive Guidance 提供的策略、指南和模式中常用的術語。若要建議項目，請使用詞彙表末尾的提供意見回饋連結。

數字

7 R

將應用程式移至雲端的七種常見遷移策略。這些策略以 Gartner 在 2011 年確定的 5 R 為基礎，包括以下內容：

- 重構/重新架構 – 充分利用雲端原生功能來移動應用程式並修改其架構，以提高敏捷性、效能和可擴展性。這通常涉及移植作業系統和資料庫。範例：將您的現場部署 Oracle 資料庫 遷移至 Amazon Aurora PostgreSQL 相容版本。
- 平台轉換 (隨即重塑) – 將應用程式移至雲端，並引入一定程度的優化以利用雲端功能。範例：將內部部署 Oracle 資料庫 遷移至 中的 Amazon Relational Database Service (Amazon RDS) for Oracle AWS 雲端。
- 重新購買 (捨棄再購買) – 切換至不同的產品，通常從傳統授權移至 SaaS 模型。範例：將您的客戶關係管理 (CRM) 系統遷移至 Salesforce.com。
- 主機轉換 (隨即轉移) – 將應用程式移至雲端，而不進行任何變更以利用雲端功能。範例：將您的現場部署 Oracle 資料庫 遷移至 中 EC2 執行個體上的 Oracle AWS 雲端。
- 重新放置 (虛擬機器監視器等級隨即轉移) – 將基礎設施移至雲端，無需購買新硬體、重寫應用程式或修改現有操作。您可以將伺服器從內部部署平台遷移到相同平台的雲端服務。範例：將 Microsoft Hyper-V應用程式 遷移至 AWS。
- 保留 (重新檢視) – 將應用程式保留在來源環境中。其中可能包括需要重要重構的應用程式，且您希望將該工作延遲到以後，以及您想要保留的舊版應用程式，因為沒有業務理由來進行遷移。
- 淘汰 – 解除委任或移除來源環境中不再需要的應用程式。

A

ABAC

請參閱屬性型存取控制。

抽象服務

請參閱 受管服務。

ACID

請參閱原子性、一致性、隔離性、耐久性。

主動-主動式遷移

一種資料庫遷移方法，其中來源和目標資料庫保持同步 (透過使用雙向複寫工具或雙重寫入操作)，且兩個資料庫都在遷移期間處理來自連接應用程式的交易。此方法支援小型、受控制批次的遷移，而不需要一次性切換。它更靈活，但需要比主動-被動遷移更多的工作。

主動-被動式遷移

一種資料庫遷移方法，其中來源和目標資料庫會保持同步，但只有來源資料庫會在資料複寫至目標資料庫時處理來自連線應用程式的交易。目標資料庫在遷移期間不接受任何交易。

彙總函數

在一組資料列上運作的 SQL 函數，會計算群組的單一傳回值。彙總函數的範例包括 SUM和 MAX。

AI

請參閱人工智慧。

AIOps

請參閱人工智慧操作。

匿名化

在資料集中永久刪除個人資訊的程序。匿名化有助於保護個人隱私權。匿名資料不再被視為個人資料。

反模式

經常用於經常性問題的解決方案，其中解決方案具有反生產力、無效或比替代解決方案更有效。

應用程式控制

一種安全方法，僅允許使用核准的應用程式，以協助保護系統免受惡意軟體攻擊。

應用程式組合

有關組織使用的每個應用程式的詳細資訊的集合，包括建置和維護應用程式的成本及其商業價值。此資訊是產品組合探索和分析程序的關鍵，有助於識別要遷移、現代化和優化的應用程式並排定其優先順序。

人工智慧 (AI)

電腦科學領域，致力於使用運算技術來執行通常與人類相關的認知功能，例如學習、解決問題和識別模式。如需詳細資訊，請參閱什麼是人工智慧？

人工智慧操作 (AIOps)

使用機器學習技術解決操作問題、減少操作事件和人工干預以及提高服務品質的程序。如需有關如何在 AWS 遷移策略中使用 AIOps 的詳細資訊，請參閱操作整合指南。

非對稱加密

一種加密演算法，它使用一對金鑰：一個用於加密的公有金鑰和一個用於解密的私有金鑰。您可以共用公有金鑰，因為它不用於解密，但對私有金鑰存取應受到高度限制。

原子性、一致性、隔離性、耐久性 (ACID)

一組軟體屬性，即使在出現錯誤、電源故障或其他問題的情況下，也能確保資料庫的資料有效性和操作可靠性。

屬性型存取控制 (ABAC)

根據使用者屬性 (例如部門、工作職責和團隊名稱) 建立精細許可的實務。如需詳細資訊，請參閱《 AWS Identity and Access Management (IAM) 文件》中的 ABAC for AWS。

授權資料來源

您存放主要版本資料的位置，被視為最可靠的資訊來源。您可以將授權資料來源中的資料複製到其他位置，以處理或修改資料，例如匿名、修訂或假名化資料。

可用區域

中的不同位置 AWS 區域 ，可隔離其他可用區域中的故障，並提供相同區域中其他可用區域的低成本、低延遲網路連線能力。

AWS 雲端採用架構 (AWS CAF)

的指導方針和最佳實務架構 AWS ，可協助組織制定高效且有效的計劃，以成功地移至雲端。 AWS CAF 將指導方針組織到六個重點領域：業務、人員、治理、平台、安全和營運。業務、人員和控管層面著重於業務技能和程序；平台、安全和操作層面著重於技術技能和程序。例如，人員層面針對處理人力資源 (HR)、人員配備功能和人員管理的利害關係人。為此， AWS CAF 為人員開發、訓練和通訊提供指引，協助組織做好成功採用雲端的準備。如需詳細資訊，請參閱 AWS CAF 網站和 AWS CAF 白皮書。

AWS 工作負載資格架構 (AWS WQF)

一種工具，可評估資料庫遷移工作負載、建議遷移策略，並提供工作預估值。 AWS WQF 隨附於 AWS Schema Conversion Tool (AWS SCT)。它會分析資料庫結構描述和程式碼物件、應用程式程式碼、相依性和效能特性，並提供評估報告。

B

錯誤的機器人

旨在中斷或傷害個人或組織的機器人。

BCP

請參閱業務持續性規劃。

行為圖

資源行為的統一互動式檢視，以及一段時間後的互動。您可以將行為圖與 Amazon Detective 搭配使用來檢查失敗的登入嘗試、可疑的 API 呼叫和類似動作。如需詳細資訊，請參閱偵測文件中的行為圖中的資料。

大端序系統

首先儲存最高有效位元組的系統。另請參閱 Endianness。

二進制分類

預測二進制結果的過程 (兩個可能的類別之一)。例如，ML 模型可能需要預測諸如「此電子郵件是否是垃圾郵件？」等問題 或「產品是書還是汽車？」

Bloom 篩選條件

一種機率性、記憶體高效的資料結構，用於測試元素是否為集的成員。

藍/綠部署

一種部署策略，您可以在其中建立兩個不同但相同的環境。您可以在一個環境 （藍色） 中執行目前的應用程式版本，並在另一個環境 （綠色） 中執行新的應用程式版本。此策略可協助您快速復原，並將影響降至最低。

機器人

透過網際網路執行自動化任務並模擬人類活動或互動的軟體應用程式。有些機器人有用或有益，例如在網際網路上編製資訊索引的 Web 爬蟲程式。某些其他機器人稱為惡意機器人，旨在中斷或傷害個人或組織。

殭屍網路

受到惡意軟體感染且受單一方控制之機器人的網路，稱為機器人繼承器或機器人運算子。殭屍網路是擴展機器人及其影響的最佳已知機制。

分支

程式碼儲存庫包含的區域。儲存庫中建立的第一個分支是主要分支。您可以從現有分支建立新分支，然後在新分支中開發功能或修正錯誤。您建立用來建立功能的分支通常稱為功能分支。當準備好發佈功能時，可以將功能分支合併回主要分支。如需詳細資訊，請參閱關於分支 (GitHub 文件)。

碎片存取

在特殊情況下，以及透過核准的程序，讓使用者快速取得他們通常無權存取 AWS 帳戶 之 的存取權。如需詳細資訊，請參閱 Well-Architected 指南中的 AWS 實作打破玻璃程序指標。

棕地策略

環境中的現有基礎設施。對系統架構採用棕地策略時，可以根據目前系統和基礎設施的限制來設計架構。如果正在擴展現有基礎設施，則可能會混合棕地和綠地策略。

緩衝快取

儲存最常存取資料的記憶體區域。

業務能力

業務如何創造價值 (例如，銷售、客戶服務或營銷)。業務能力可驅動微服務架構和開發決策。如需詳細資訊，請參閱在 AWS上執行容器化微服務白皮書的圍繞業務能力進行組織部分。

業務連續性規劃 (BCP)

一種解決破壞性事件 (如大規模遷移) 對營運的潛在影響並使業務能夠快速恢復營運的計畫。

C

CAF

請參閱AWS 雲端採用架構。

Canary 部署

版本對最終使用者的緩慢和增量版本。當您有信心時，您可以部署新版本並完全取代目前的版本。

CCoE

請參閱 Cloud Center of Excellence。

CDC

請參閱變更資料擷取。

變更資料擷取 (CDC)

追蹤對資料來源 (例如資料庫表格) 的變更並記錄有關變更的中繼資料的程序。您可以將 CDC 用於各種用途，例如稽核或複寫目標系統中的變更以保持同步。

混沌工程

故意引入故障或破壞性事件，以測試系統的彈性。您可以使用 AWS Fault Injection Service (AWS FIS) 來執行實驗，為您的 AWS 工作負載帶來壓力，並評估其回應。

CI/CD

請參閱持續整合和持續交付。

分類

有助於產生預測的分類程序。用於分類問題的 ML 模型可預測離散值。離散值永遠彼此不同。例如，模型可能需要評估影像中是否有汽車。

用戶端加密

在目標 AWS 服務 接收資料之前，在本機加密資料。

雲端卓越中心 (CCoE)

一個多學科團隊，可推動整個組織的雲端採用工作，包括開發雲端最佳實務、調動資源、制定遷移時間表以及領導組織進行大規模轉型。如需詳細資訊，請參閱 AWS 雲端 企業策略部落格上的 CCoE 文章。

雲端運算

通常用於遠端資料儲存和 IoT 裝置管理的雲端技術。雲端運算通常連接到邊緣運算技術。

雲端操作模型

在 IT 組織中，用於建置、成熟和最佳化一或多個雲端環境的操作模型。如需詳細資訊，請參閱建置您的雲端操作模型。

採用雲端階段

組織在遷移至 時通常會經歷的四個階段 AWS 雲端：

- 專案 – 執行一些與雲端相關的專案以進行概念驗證和學習用途
- 基礎 – 進行基礎投資以擴展雲端採用 (例如，建立登陸區域、定義 CCoE、建立營運模型)

- 遷移 – 遷移個別應用程式
- 重塑 – 優化產品和服務，並在雲端中創新

這些階段由 Stephen Orban 在部落格文章 The Journey Toward Cloud-First 和企業策略部落格上的採用階段中定義。 AWS 雲端 如需有關它們如何與 AWS 遷移策略關聯的資訊，請參閱遷移整備指南。

CMDB

請參閱組態管理資料庫。

程式碼儲存庫

透過版本控制程序來儲存及更新原始程式碼和其他資產 (例如文件、範例和指令碼) 的位置。常見的雲端儲存庫包括 GitHub或 Bitbucket Cloud。程式碼的每個版本都稱為分支。在微服務結構中，每個儲存庫都專用於單個功能。單一 CI/CD 管道可以使用多個儲存庫。

冷快取

一種緩衝快取，它是空的、未填充的，或者包含過時或不相關的資料。這會影響效能，因為資料庫執行個體必須從主記憶體或磁碟讀取，這比從緩衝快取讀取更慢。

冷資料

很少存取且通常是歷史資料的資料。查詢這類資料時，通常可接受慢查詢。將此資料移至效能較低且成本較低的儲存層或類別，可以降低成本。

電腦視覺 (CV)

使用機器學習從數位影像和影片等視覺化格式分析和擷取資訊的 AI 欄位。例如，Amazon SageMaker AI 提供 CV 的影像處理演算法。

組態偏離

對於工作負載，組態會從預期狀態變更。這可能會導致工作負載變得不合規，而且通常是漸進和無意的。

組態管理資料庫 (CMDB)

儲存和管理有關資料庫及其 IT 環境的資訊的儲存庫，同時包括硬體和軟體元件及其組態。您通常在遷移的產品組合探索和分析階段使用 CMDB 中的資料。

一致性套件

您可以組合的 AWS Config 規則和修補動作集合，以自訂您的合規和安全檢查。您可以使用 YAML 範本，將一致性套件部署為 AWS 帳戶 和 區域中或整個組織的單一實體。如需詳細資訊，請參閱 AWS Config 文件中的一致性套件。

持續整合和持續交付 (CI/CD)

自動化軟體發行程序的來源、建置、測試、暫存和生產階段的程序。CI/CD 通常被描述為管道。CI/CD 可協助您將程序自動化、提升生產力、改善程式碼品質以及加快交付速度。如需詳細資訊，請參閱持續交付的優點。CD 也可表示持續部署。如需詳細資訊，請參閱持續交付與持續部署。

CV

請參閱電腦視覺。

D

靜態資料

網路中靜止的資料，例如儲存中的資料。

資料分類

根據重要性和敏感性來識別和分類網路資料的程序。它是所有網路安全風險管理策略的關鍵組成部分，因為它可以協助您確定適當的資料保護和保留控制。資料分類是 AWS Well-Architected Framework 中安全支柱的元件。如需詳細資訊，請參閱資料分類。

資料偏離

生產資料與用於訓練 ML 模型的資料之間有意義的變化，或輸入資料隨時間有意義的變更。資料偏離可以降低 ML 模型預測的整體品質、準確性和公平性。

傳輸中的資料

在您的網路中主動移動的資料，例如在網路資源之間移動。

資料網格

架構架構，提供分散式、分散式資料擁有權與集中式管理。

資料最小化

僅收集和處理嚴格必要資料的原則。在 中實作資料最小化 AWS 雲端 可以降低隱私權風險、成本和分析碳足跡。

資料周邊

AWS 環境中的一組預防性防護機制，可協助確保只有信任的身分才能從預期的網路存取信任的資源。如需詳細資訊，請參閱在 上建置資料周邊 AWS。

資料預先處理

將原始資料轉換成 ML 模型可輕鬆剖析的格式。預處理資料可能意味著移除某些欄或列，並解決遺失、不一致或重複的值。

資料來源

在整個資料生命週期中追蹤資料的來源和歷史記錄的程序，例如資料的產生、傳輸和儲存方式。

資料主體

正在收集和處理資料的個人。

資料倉儲

支援商業智慧的資料管理系統，例如 分析。資料倉儲通常包含大量歷史資料，通常用於查詢和分析。

資料庫定義語言 (DDL)

用於建立或修改資料庫中資料表和物件之結構的陳述式或命令。

資料庫處理語言 (DML)

用於修改 (插入、更新和刪除) 資料庫中資訊的陳述式或命令。

DDL

請參閱資料庫定義語言。

深度整體

結合多個深度學習模型進行預測。可以使用深度整體來獲得更準確的預測或估計預測中的不確定性。

深度學習

一個機器學習子領域，它使用多層人工神經網路來識別感興趣的輸入資料與目標變數之間的對應關係。

深度防禦

這是一種資訊安全方法，其中一系列的安全機制和控制項會在整個電腦網路中精心分層，以保護網路和其中資料的機密性、完整性和可用性。當您在 上採用此策略時 AWS，您可以在 AWS Organizations 結構的不同層新增多個控制項，以協助保護資源。例如，defense-in-depth方法可能會結合多重要素驗證、網路分割和加密。

委派的管理員

在 中 AWS Organizations，相容的服務可以註冊 AWS 成員帳戶來管理組織的帳戶，並管理該服務的許可。此帳戶稱為該服務的委派管理員。如需詳細資訊和相容服務清單，請參閱 AWS Organizations 文件中的可搭配 AWS Organizations運作的服務。

deployment

在目標環境中提供應用程式、新功能或程式碼修正的程序。部署涉及在程式碼庫中實作變更，然後在應用程式環境中建置和執行該程式碼庫。

開發環境

請參閱 環境。

偵測性控制

一種安全控制，用於在事件發生後偵測、記錄和提醒。這些控制是第二道防線，提醒您注意繞過現有預防性控制的安全事件。如需詳細資訊，請參閱在 AWS上實作安全控制中的偵測性控制。

開發值串流映射 (DVSM)

一種程序，用於識別並優先考慮對軟體開發生命週期中的速度和品質造成負面影響的限制。DVSM 擴展了最初專為精簡製造實務設計的價值串流映射程序。它著重於透過軟體開發程序建立和移動價值所需的步驟和團隊。

數位分身

真實世界系統的虛擬呈現，例如建築物、工廠、工業設備或生產線。數位分身支援預測性維護、遠端監控和生產最佳化。

維度資料表

在星星結構描述中，較小的資料表包含有關事實資料表中量化資料的資料屬性。維度資料表屬性通常是文字欄位或離散數字，其行為類似於文字。這些屬性通常用於查詢限制、篩選和結果集標記。

災難

防止工作負載或系統在其主要部署位置實現其業務目標的事件。這些事件可能是自然災難、技術故障或人為動作的結果，例如意外設定錯誤或惡意軟體攻擊。

災難復原 (DR)

您用來將災難造成的停機時間和資料遺失降至最低的策略和程序。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的 上工作負載的災難復原 AWS：雲端中的復原。

DML

請參閱資料庫處理語言。

領域驅動的設計

一種開發複雜軟體系統的方法，它會將其元件與每個元件所服務的不斷發展的領域或核心業務目標相關聯。Eric Evans 在其著作 Domain-Driven Design: Tackling Complexity in the Heart of Software (Boston: Addison-Wesley Professional, 2003) 中介紹了這一概念。如需有關如何將領域驅動的設計與 strangler fig 模式搭配使用的資訊，請參閱使用容器和 Amazon API Gateway 逐步現代化舊版 Microsoft ASP.NET (ASMX) Web 服務。

DR

請參閱災難復原。

偏離偵測

追蹤與基準組態的偏差。例如，您可以使用 AWS CloudFormation 來偵測系統資源中的偏離，也可以使用 AWS Control Tower 來偵測登陸區域中可能影響控管要求合規性的變更。 https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-stack-drift.html

DVSM

請參閱開發值串流映射。

E

EDA

請參閱探索性資料分析。

EDI

請參閱電子資料交換。

邊緣運算

提升 IoT 網路邊緣智慧型裝置運算能力的技術。與雲端運算相比，邊緣運算可以減少通訊延遲並改善回應時間。

電子資料交換 (EDI)

在組織之間自動交換商業文件。如需詳細資訊，請參閱什麼是電子資料交換。

加密

將人類可讀取的純文字資料轉換為加密文字的運算程序。

加密金鑰

由加密演算法產生的隨機位元的加密字串。金鑰長度可能有所不同，每個金鑰的設計都是不可預測且唯一的。

端序

位元組在電腦記憶體中的儲存順序。大端序系統首先儲存最高有效位元組。小端序系統首先儲存最低有效位元組。

端點

請參閱 服務端點。

端點服務

您可以在虛擬私有雲端 (VPC) 中託管以與其他使用者共用的服務。您可以使用 建立端點服務，AWS PrivateLink 並將許可授予其他 AWS 帳戶 或 AWS Identity and Access Management (IAM) 委託人。這些帳戶或主體可以透過建立介面 VPC 端點私下連接至您的端點服務。如需詳細資訊，請參閱 Amazon Virtual Private Cloud (Amazon VPC) 文件中的建立端點服務。

企業資源規劃 (ERP)

一種系統，可自動化和管理企業的關鍵業務流程 （例如會計、MES 和專案管理）。

信封加密

使用另一個加密金鑰對某個加密金鑰進行加密的程序。如需詳細資訊，請參閱 () 文件中的信封加密。 AWS Key Management Service AWS KMS

環境

執行中應用程式的執行個體。以下是雲端運算中常見的環境類型：

- 開發環境 – 執行中應用程式的執行個體，只有負責維護應用程式的核心團隊才能使用。開發環境用來測試變更，然後再將開發環境提升到較高的環境。此類型的環境有時稱為測試環境。
- 較低的環境 – 應用程式的所有開發環境，例如用於初始建置和測試的開發環境。
- 生產環境 – 最終使用者可以存取的執行中應用程式的執行個體。在 CI/CD 管道中，生產環境是最後一個部署環境。
- 較高的環境 – 核心開發團隊以外的使用者可存取的所有環境。這可能包括生產環境、生產前環境以及用於使用者接受度測試的環境。

epic

在敏捷方法中，有助於組織工作並排定工作優先順序的功能類別。epic 提供要求和實作任務的高層級描述。例如， AWS CAF 安全概念包括身分和存取管理、偵測控制、基礎設施安全、資料保護和事件回應。如需有關 AWS 遷移策略中的 Epic 的詳細資訊，請參閱計畫實作指南。

ERP

請參閱企業資源規劃。

探索性資料分析 (EDA)

分析資料集以了解其主要特性的過程。您收集或彙總資料，然後執行初步調查以尋找模式、偵測異常並檢查假設。透過計算摘要統計並建立資料可視化來執行 EDA。

F

事實資料表

星狀結構描述中的中央資料表。它存放有關業務操作的量化資料。一般而言，事實資料表包含兩種類型的資料欄：包含度量的資料，以及包含維度資料表外部索引鍵的資料欄。

快速失敗

一種使用頻繁且增量測試來縮短開發生命週期的理念。這是敏捷方法的關鍵部分。

故障隔離界限

在 中 AWS 雲端，像是可用區域 AWS 區域、控制平面或資料平面等界限會限制故障的影響，並有助於改善工作負載的彈性。如需詳細資訊，請參閱AWS 故障隔離界限。

功能分支

請參閱分支。

特徵

用來進行預測的輸入資料。例如，在製造環境中，特徵可能是定期從製造生產線擷取的影像。

功能重要性

特徵對於模型的預測有多重要。這通常表示為可以透過各種技術來計算的數值得分，例如 Shapley Additive Explanations (SHAP) 和積分梯度。如需詳細資訊，請參閱 的機器學習模型可解譯性 AWS。

特徵轉換

優化 ML 程序的資料，包括使用其他來源豐富資料、調整值、或從單一資料欄位擷取多組資訊。這可讓 ML 模型從資料中受益。例如，如果將「2021-05-27 00:15:37」日期劃分為「2021」、「五月」、「週四」和「15」，則可以協助學習演算法學習與不同資料元件相關聯的細微模式。

少量擷取提示

在要求 LLM 執行類似的任務之前，提供少量示範任務和所需輸出的範例。此技術是內容內學習的應用程式，其中模型會從內嵌在提示中的範例 (快照) 中學習。少量的提示對於需要特定格式、推理或網域知識的任務來說非常有效。另請參閱零鏡頭提示。

FGAC

請參閱精細存取控制。

精細存取控制 (FGAC)

使用多個條件來允許或拒絕存取請求。

閃切遷移

一種資料庫遷移方法，透過變更資料擷取使用連續資料複寫，以盡可能在最短的時間內遷移資料，而不是使用分階段方法。目標是將停機時間降至最低。

FM

請參閱基礎模型。

基礎模型 (FM)

大型深度學習神經網路，已在廣義和未標記資料的大量資料集上進行訓練。FMs 能夠執行各種一般任務，例如了解語言、產生文字和影像，以及以自然語言交談。如需詳細資訊，請參閱什麼是基礎模型。

G

生成式 AI

已針對大量資料進行訓練的 AI 模型子集，可使用簡單的文字提示建立新的內容和成品，例如影像、影片、文字和音訊。如需詳細資訊，請參閱什麼是生成式 AI。

地理封鎖

請參閱地理限制。

地理限制 (地理封鎖)

Amazon CloudFront 中的選項，可防止特定國家/地區的使用者存取內容分發。您可以使用允許清單或封鎖清單來指定核准和禁止的國家/地區。如需詳細資訊，請參閱 CloudFront 文件中的限制內容的地理分佈。

Gitflow 工作流程

這是一種方法，其中較低和較高環境在原始碼儲存庫中使用不同分支。Gitflow 工作流程被視為舊版，而以幹線為基礎的工作流程是現代、偏好的方法。

黃金影像

系統或軟體的快照，做為部署該系統或軟體新執行個體的範本。例如，在製造中，黃金映像可用於在多個裝置上佈建軟體，並有助於提高裝置製造操作的速度、可擴展性和生產力。

綠地策略

新環境中缺乏現有基礎設施。對系統架構採用綠地策略時，可以選擇所有新技術，而不會限制與現有基礎設施的相容性，也稱為棕地。如果正在擴展現有基礎設施，則可能會混合棕地和綠地策略。

防護機制

有助於跨組織單位 (OU) 來管控資源、政策和合規的高層級規則。預防性防護機制會強制執行政策，以確保符合合規標準。透過使用服務控制政策和 IAM 許可界限來將其實作。偵測性防護機制可偵測政策違規和合規問題，並產生提醒以便修正。它們是透過使用 AWS Config AWS Security Hub CSPM、Amazon GuardDuty、、Amazon Inspector AWS Trusted Advisor和自訂 AWS Lambda 檢查來實作。

H

HA

請參閱高可用性。

異質資料庫遷移

將來源資料庫遷移至使用不同資料庫引擎的目標資料庫 (例如，Oracle 至 Amazon Aurora)。異質遷移通常是重新架構工作的一部分，而轉換結構描述可能是一項複雜任務。AWS 提供有助於結構描述轉換的 AWS SCT。

高可用性 (HA)

在遇到挑戰或災難時，工作負載能夠在不介入的情況下持續運作。HA 系統的設計目的是自動容錯移轉、持續提供高品質的效能，以及處理不同的負載和故障，並將效能影響降至最低。

歷史現代化

一種方法，用於現代化和升級操作技術 (OT) 系統，以更好地滿足製造業的需求。歷史資料是一種資料庫，用於從工廠中的各種來源收集和存放資料。

保留資料

從用於訓練機器學習模型的資料集中保留的部分歷史標記資料。您可以使用保留資料，透過比較模型預測與保留資料來評估模型效能。

異質資料庫遷移

將您的來源資料庫遷移至共用相同資料庫引擎的目標資料庫 (例如，Microsoft SQL Server 至 Amazon RDS for SQL Server)。同質遷移通常是主機轉換或平台轉換工作的一部分。您可以使用原生資料庫公用程式來遷移結構描述。

熱資料

經常存取的資料，例如即時資料或最近的轉譯資料。此資料通常需要高效能儲存層或類別，才能提供快速的查詢回應。

修補程序

緊急修正生產環境中的關鍵問題。由於其緊迫性，通常會在典型 DevOps 發行工作流程之外執行修補程式。

超級護理期間

在切換後，遷移團隊在雲端管理和監控遷移的應用程式以解決任何問題的時段。通常，此期間的長度為 1-4 天。在超級護理期間結束時，遷移團隊通常會將應用程式的責任轉移給雲端營運團隊。

I

IaC

將基礎設施視為程式碼。

身分型政策

連接至一或多個 IAM 主體的政策，可定義其在 AWS 雲端 環境中的許可。

閒置應用程式

90 天期間 CPU 和記憶體平均使用率在 5% 至 20% 之間的應用程式。在遷移專案中，通常會淘汰這些應用程式或將其保留在內部部署。

IIoT

請參閱工業物聯網。

不可變的基礎設施

為生產工作負載部署新基礎設施的模型，而不是更新、修補或修改現有的基礎設施。不可變基礎設施本質上比可變基礎設施更一致、可靠且可預測。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的使用不可變基礎設施的部署最佳實務。

傳入 (輸入) VPC

在 AWS 多帳戶架構中，接受、檢查和路由來自應用程式外部之網路連線的 VPC。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

增量遷移

一種切換策略，您可以在其中將應用程式分成小部分遷移，而不是執行單一、完整的切換。例如，您最初可能只將一些微服務或使用者移至新系統。確認所有項目都正常運作之後，您可以逐步移動其他微服務或使用者，直到可以解除委任舊式系統。此策略可降低與大型遷移關聯的風險。

工業 4.0

由 Klaus Schwab 於 2016 年推出的術語，透過連線能力、即時資料、自動化、分析和 AI/ML 的進展，指製造程序的現代化。

基礎設施

應用程式環境中包含的所有資源和資產。

基礎設施即程式碼 (IaC)

透過一組組態檔案來佈建和管理應用程式基礎設施的程序。IaC 旨在協助您集中管理基礎設施，標準化資源並快速擴展，以便新環境可重複、可靠且一致。

工業物聯網 (IIoT)

在製造業、能源、汽車、醫療保健、生命科學和農業等產業領域使用網際網路連線的感測器和裝置。如需詳細資訊，請參閱建立工業物聯網 (IIoT) 數位轉型策略。

檢查 VPC

在 AWS 多帳戶架構中，集中式 VPC 可管理 VPCs 之間 （在相同或不同的 中 AWS 區域)、網際網路和內部部署網路之間的網路流量檢查。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

物聯網 (IoT)

具有內嵌式感測器或處理器的相連實體物體網路，其透過網際網路或本地通訊網路與其他裝置和系統進行通訊。如需詳細資訊，請參閱什麼是 IoT？

可解釋性

機器學習模型的一個特徵，描述了人類能夠理解模型的預測如何依賴於其輸入的程度。如需詳細資訊，請參閱 的機器學習模型可解譯性 AWS。

IoT

請參閱物聯網。

IT 資訊庫 (ITIL)

一組用於交付 IT 服務並使這些服務與業務需求保持一致的最佳實務。ITIL 為 ITSM 提供了基礎。

IT 服務管理 (ITSM)

與組織的設計、實作、管理和支援 IT 服務關聯的活動。如需有關將雲端操作與 ITSM 工具整合的資訊，請參閱操作整合指南。

ITIL

請參閱 IT 資訊庫。

ITSM

請參閱 IT 服務管理。

L

標籤型存取控制 (LBAC)

強制存取控制 (MAC) 的實作，其中使用者和資料本身都會獲得明確指派的安全標籤值。使用者安全標籤和資料安全標籤之間的交集會決定使用者可以看到哪些資料列和資料欄。

登陸區域

登陸區域是架構良好的多帳戶 AWS 環境，可擴展且安全。這是一個起點，您的組織可以從此起點快速啟動和部署工作負載與應用程式，並對其安全和基礎設施環境充滿信心。如需有關登陸區域的詳細資訊，請參閱設定安全且可擴展的多帳戶 AWS 環境。

大型語言模型 (LLM)

預先訓練大量資料的深度學習 AI 模型。LLM 可以執行多個任務，例如回答問題、摘要文件、將文字翻譯成其他語言，以及完成句子。如需詳細資訊，請參閱什麼是 LLMs。

大型遷移

遷移 300 部或更多伺服器。

LBAC

請參閱標籤型存取控制。

最低權限

授予執行任務所需之最低許可的安全最佳實務。如需詳細資訊，請參閱 IAM 文件中的套用最低權限許可。

隨即轉移

請參閱 7 個 R。

小端序系統

首先儲存最低有效位元組的系統。另請參閱 Endianness。

LLM

請參閱大型語言模型。

較低的環境

請參閱 環境。

M

機器學習 (ML)

一種使用演算法和技術進行模式識別和學習的人工智慧。機器學習會進行分析並從記錄的資料 (例如物聯網 (IoT) 資料) 中學習，以根據模式產生統計模型。如需詳細資訊，請參閱機器學習。

主要分支

請參閱分支。

惡意軟體

旨在危及電腦安全或隱私權的軟體。惡意軟體可能會中斷電腦系統、洩露敏感資訊，或取得未經授權的存取。惡意軟體的範例包括病毒、蠕蟲、勒索軟體、特洛伊木馬程式、間諜軟體和鍵盤記錄器。

受管服務

AWS 服務 會 AWS 操作基礎設施層、作業系統和平台，而您會存取端點來存放和擷取資料。Amazon Simple Storage Service (Amazon S3) 和 Amazon DynamoDB 是受管服務的範例。這些也稱為抽象服務。

製造執行系統 (MES)

一種軟體系統，用於追蹤、監控、記錄和控制生產程序，將原物料轉換為現場成品。

MAP

請參閱遷移加速計劃。

機制

建立工具、推動工具採用，然後檢查結果以進行調整的完整程序。機制是在操作時強化和改善自身的循環。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的建置機制。

成員帳戶

除了屬於組織一部分的管理帳戶 AWS 帳戶 之外的所有 AWS Organizations。帳戶一次只能是一個組織的成員。

製造執行系統

請參閱製造執行系統。

訊息佇列遙測傳輸 (MQTT)

根據發佈/訂閱模式的輕量型machine-to-machine(M2M) 通訊協定，適用於資源受限的 IoT 裝置。

微服務

一種小型的獨立服務，它可透過定義明確的 API 進行通訊，通常由小型獨立團隊擁有。例如，保險系統可能包含對應至業務能力 (例如銷售或行銷) 或子領域 (例如購買、索賠或分析) 的微服務。微服務的優點包括靈活性、彈性擴展、輕鬆部署、可重複使用的程式碼和適應力。如需詳細資訊，請參閱使用無 AWS 伺服器服務整合微服務。

微服務架構

一種使用獨立元件來建置應用程式的方法，這些元件會以微服務形式執行每個應用程式程序。這些微服務會使用輕量型 API，透過明確定義的介面進行通訊。此架構中的每個微服務都可以進行更新、部署和擴展，以滿足應用程式特定功能的需求。如需詳細資訊，請參閱實作微服務 AWS。

Migration Acceleration Program (MAP)

此 AWS 計畫提供諮詢支援、訓練和服務，以協助組織建立強大的營運基礎，以移至雲端，並協助抵銷遷移的初始成本。MAP 包括用於有條不紊地執行舊式遷移的遷移方法以及一組用於自動化和加速常見遷移案例的工具。

大規模遷移

將大部分應用程式組合依波次移至雲端的程序，在每個波次中，都會以更快的速度移動更多應用程式。此階段使用從早期階段學到的最佳實務和經驗教訓來實作團隊、工具和流程的遷移工廠，以透過自動化和敏捷交付簡化工作負載的遷移。這是 AWS 遷移策略的第三階段。

遷移工廠

可透過自動化、敏捷的方法簡化工作負載遷移的跨職能團隊。遷移工廠團隊通常包括營運、業務分析師和擁有者、遷移工程師、開發人員以及從事 Sprint 工作的 DevOps 專業人員。20% 至 50% 之間的企業應用程式組合包含可透過工廠方法優化的重複模式。如需詳細資訊，請參閱此內容集中的遷移工廠的討論和雲端遷移工廠指南。

遷移中繼資料

有關完成遷移所需的應用程式和伺服器的資訊。每種遷移模式都需要一組不同的遷移中繼資料。遷移中繼資料的範例包括目標子網路、安全群組和 AWS 帳戶。

遷移模式

可重複的遷移任務，詳細描述遷移策略、遷移目的地以及所使用的遷移應用程式或服務。範例：使用 AWS Application Migration Service 重新託管遷移至 Amazon EC2。

遷移組合評定 (MPA)

線上工具，提供驗證商業案例以遷移至 的資訊 AWS 雲端。MPA 提供詳細的組合評定 (伺服器適當規模、定價、總體擁有成本比較、遷移成本分析) 以及遷移規劃 (應用程式資料分析和資料收集、應用程式分組、遷移優先順序，以及波次規劃)。MPA 工具 （需要登入） 可供所有 AWS 顧問和 APN 合作夥伴顧問免費使用。

遷移準備程度評定 (MRA)

使用 AWS CAF 取得組織雲端整備狀態的洞見、識別優缺點，以及建立行動計劃以消除已識別差距的程序。如需詳細資訊，請參閱遷移準備程度指南。MRA 是 AWS 遷移策略的第一階段。

遷移策略

用來將工作負載遷移至 的方法 AWS 雲端。如需詳細資訊，請參閱本詞彙表中的 7 個 Rs 項目，並請參閱動員您的組織以加速大規模遷移。

機器學習 (ML)

請參閱機器學習。

現代化

將過時的 (舊版或單一) 應用程式及其基礎架構轉換為雲端中靈活、富有彈性且高度可用的系統，以降低成本、提高效率並充分利用創新。如需詳細資訊，請參閱《》中的現代化應用程式的策略 AWS 雲端。

現代化準備程度評定

這項評估可協助判斷組織應用程式的現代化準備程度；識別優點、風險和相依性；並確定組織能夠在多大程度上支援這些應用程式的未來狀態。評定的結果就是目標架構的藍圖、詳細說明現代化程序的開發階段和里程碑的路線圖、以及解決已發現的差距之行動計畫。如需詳細資訊，請參閱《》中的評估應用程式的現代化準備 AWS 雲端程度。

單一應用程式 (單一)

透過緊密結合的程序作為單一服務執行的應用程式。單一應用程式有幾個缺點。如果一個應用程式功能遇到需求激增，則必須擴展整個架構。當程式碼庫增長時，新增或改進單一應用程式的功能也會變得更加複雜。若要解決這些問題，可以使用微服務架構。如需詳細資訊，請參閱將單一體系分解為微服務。

MPA

請參閱遷移產品組合評估。

MQTT

請參閱訊息佇列遙測傳輸。

多類別分類

一個有助於產生多類別預測的過程 (預測兩個以上的結果之一)。例如，機器學習模型可能會詢問「此產品是書籍、汽車還是電話？」 或者「這個客戶對哪種產品類別最感興趣？」

可變基礎設施

更新和修改生產工作負載現有基礎設施的模型。為了提高一致性、可靠性和可預測性， AWS Well-Architected Framework 建議使用不可變基礎設施做為最佳實務。

O

OAC

請參閱原始存取控制。

OAI

請參閱原始存取身分。

OCM

請參閱組織變更管理。

離線遷移

一種遷移方法，可在遷移過程中刪除來源工作負載。此方法涉及延長停機時間，通常用於小型非關鍵工作負載。

OI

請參閱 操作整合。

OLA

請參閱操作層級協議。

線上遷移

一種遷移方法，無需離線即可將來源工作負載複製到目標系統。連接至工作負載的應用程式可在遷移期間繼續運作。此方法涉及零至最短停機時間，通常用於關鍵的生產工作負載。

OPC-UA

請參閱開啟程序通訊 - 統一架構。

開放程序通訊 - 統一架構 (OPC-UA)

用於工業自動化machine-to-machine(M2M) 通訊協定。OPC-UA 提供資料加密、身分驗證和授權機制的互通性標準。

操作水準協議 (OLA)

一份協議，闡明 IT 職能群組承諾向彼此提供的內容，以支援服務水準協議 (SLA)。

操作整備審查 (ORR)

問題和相關最佳實務的檢查清單，可協助您了解、評估、預防或減少事件和可能失敗的範圍。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的操作準備度審查 (ORR)。

操作技術 (OT)

使用實體環境控制工業操作、設備和基礎設施的硬體和軟體系統。在製造中，OT 和資訊技術 (IT) 系統的整合是工業 4.0 轉型的關鍵重點。

操作整合 (OI)

在雲端中將操作現代化的程序，其中包括準備程度規劃、自動化和整合。如需詳細資訊，請參閱操作整合指南。

組織追蹤

建立的線索 AWS CloudTrail 會記錄 AWS 帳戶 組織中所有 的所有事件 AWS Organizations。在屬於組織的每個 AWS 帳戶 中建立此追蹤，它會跟蹤每個帳戶中的活動。如需詳細資訊，請參閱 CloudTrail 文件中的建立組織追蹤。

組織變更管理 (OCM)

用於從人員、文化和領導力層面管理重大、顛覆性業務轉型的架構。OCM 透過加速變更採用、解決過渡問題，以及推動文化和組織變更，協助組織為新系統和策略做好準備，並轉移至新系統和策略。在 AWS 遷移策略中，此架構稱為人員加速，因為雲端採用專案所需的變更速度。如需詳細資訊，請參閱 OCM 指南。

原始存取控制 (OAC)

CloudFront 中的增強型選項，用於限制存取以保護 Amazon Simple Storage Service (Amazon S3) 內容。OAC 支援所有 S3 儲存貯體中的所有伺服器端加密 AWS KMS (SSE-KMS) AWS 區域，以及對 S3 儲存貯體的動態PUT和DELETE請求。

原始存取身分 (OAI)

CloudFront 中的一個選項，用於限制存取以保護 Amazon S3 內容。當您使用 OAI 時，CloudFront 會建立一個可供 Amazon S3 進行驗證的主體。經驗證的主體只能透過特定 CloudFront 分發來存取 S3 儲存貯體中的內容。另請參閱 OAC，它可提供更精細且增強的存取控制。

ORR

請參閱操作整備審核。

OT

請參閱操作技術。

傳出 (輸出) VPC

在 AWS 多帳戶架構中，處理從應用程式內啟動之網路連線的 VPC。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

P

許可界限

附接至 IAM 主體的 IAM 管理政策，可設定使用者或角色擁有的最大許可。如需詳細資訊，請參閱 IAM 文件中的許可界限。

個人身分識別資訊 (PII)

當直接檢視或與其他相關資料配對時，可用來合理推斷個人身分的資訊。PII 的範例包括名稱、地址和聯絡資訊。

PII

請參閱個人身分識別資訊。

手冊

一組預先定義的步驟，可擷取與遷移關聯的工作，例如在雲端中提供核心操作功能。手冊可以採用指令碼、自動化執行手冊或操作現代化環境所需的程序或步驟摘要的形式。

PLC

請參閱可程式設計邏輯控制器。

PLM

請參閱產品生命週期管理。

政策

可定義許可的物件（請參閱身分型政策）、指定存取條件（請參閱資源型政策），或定義組織中所有帳戶的最大許可 AWS Organizations（請參閱服務控制政策）。

混合持久性

根據資料存取模式和其他需求，獨立選擇微服務的資料儲存技術。如果您的微服務具有相同的資料儲存技術，則其可能會遇到實作挑戰或效能不佳。如果微服務使用最適合其需求的資料儲存，則可以更輕鬆地實作並達到更好的效能和可擴展性。

組合評定

探索、分析應用程式組合並排定其優先順序以規劃遷移的程序。如需詳細資訊，請參閱評估遷移準備程度。

述詞

傳回 `true`或 的查詢條件`false`，通常位於 `WHERE`子句中。

述詞下推

一種資料庫查詢最佳化技術，可在傳輸前篩選查詢中的資料。這可減少必須從關聯式資料庫擷取和處理的資料量，並改善查詢效能。

預防性控制

旨在防止事件發生的安全控制。這些控制是第一道防線，可協助防止對網路的未經授權存取或不必要變更。如需詳細資訊，請參閱在 AWS上實作安全控制中的預防性控制。

委託人

中可執行動作和存取資源 AWS 的實體。此實體通常是 AWS 帳戶、IAM 角色或使用者的根使用者。如需詳細資訊，請參閱 IAM 文件中角色術語和概念中的主體。

依設計的隱私權

透過整個開發程序將隱私權納入考量的系統工程方法。

私有託管區域

一種容器，它包含有關您希望 Amazon Route 53 如何回應一個或多個 VPC 內的域及其子域之 DNS 查詢的資訊。如需詳細資訊，請參閱 Route 53 文件中的使用私有託管區域。

主動控制

旨在防止部署不合規資源的安全控制。這些控制項會在佈建資源之前對其進行掃描。如果資源不符合控制項，則不會佈建。如需詳細資訊，請參閱 AWS Control Tower 文件中的控制項參考指南，並參閱實作安全控制項中的主動控制項。 AWS

產品生命週期管理 (PLM)

管理產品整個生命週期的資料和程序，從設計、開發和啟動，到成長和成熟，再到拒絕和移除。

生產環境

請參閱 環境。

可程式設計邏輯控制器 (PLC)

在製造中，高度可靠、可調整的電腦，可監控機器並自動化製造程序。

提示鏈結

使用一個 LLM 提示的輸出做為下一個提示的輸入，以產生更好的回應。此技術用於將複雜任務分解為子任務，或反覆精簡或展開初步回應。它有助於提高模型回應的準確性和相關性，並允許更精細、個人化的結果。

擬匿名化

以預留位置值取代資料集中個人識別符的程序。假名化有助於保護個人隱私權。假名化資料仍被視為個人資料。

發佈/訂閱 (pub/sub)

一種模式，可啟用微服務之間的非同步通訊，以提高可擴展性和回應能力。例如，在微服務型 MES 中，微服務可以將事件訊息發佈到其他微服務可訂閱的頻道。系統可以新增新的微服務，而無需變更發佈服務。

Q

查詢計劃

一系列步驟，如指示，用於存取 SQL 關聯式資料庫系統中的資料。

查詢計劃迴歸

在資料庫服務優化工具選擇的計畫比對資料庫環境進行指定的變更之前的計畫不太理想時。這可能因為對統計資料、限制條件、環境設定、查詢參數繫結的變更以及資料庫引擎的更新所導致。

R

RACI 矩陣

請參閱負責、負責、諮詢、告知 (RACI)。

RAG

請參閱擷取增強產生。

勒索軟體

一種惡意軟體，旨在阻止對計算機系統或資料的存取，直到付款為止。

RASCI 矩陣

請參閱負責、負責、諮詢、告知 (RACI)。

RCAC

請參閱資料列和資料欄存取控制。

僅供讀取複本

用於唯讀用途的資料庫複本。您可以將查詢路由至僅供讀取複本以減少主資料庫的負載。

重新架構師

請參閱 7 Rs。

復原點目標 (RPO)

自上次資料復原點以來可接受的時間上限。這會決定最後一個復原點與服務中斷之間可接受的資料遺失。

復原時間目標 (RTO)

服務中斷與服務還原之間的可接受延遲上限。

重構

請參閱 7 Rs。

區域

地理區域中的 AWS 資源集合。每個 AWS 區域 都獨立於其他 ，以提供容錯能力、穩定性和彈性。如需詳細資訊，請參閱指定 AWS 區域 您的帳戶可以使用哪些。

迴歸

預測數值的 ML 技術。例如，為了解決「這房子會賣什麼價格？」的問題 ML 模型可以使用線性迴歸模型，根據已知的房屋事實 (例如，平方英尺) 來預測房屋的銷售價格。

重新託管

請參閱 7 Rs。

版本

在部署程序中，它是將變更提升至生產環境的動作。

重新定位

請參閱 7 個 R。

Replatform

請參閱 7 個 R。

回購

請參閱 7 Rs。

彈性

應用程式抵禦中斷或從中斷中復原的能力。在 中規劃彈性時，高可用性和災難復原是常見的考量 AWS 雲端。如需詳細資訊，請參閱AWS 雲端 彈性。

資源型政策

附接至資源的政策，例如 Amazon S3 儲存貯體、端點或加密金鑰。這種類型的政策會指定允許存取哪些主體、支援的動作以及必須滿足的任何其他條件。

負責者、當責者、事先諮詢者和事後告知者 (RACI) 矩陣

定義所有涉及遷移活動和雲端操作之各方的角色和責任的矩陣。矩陣名稱衍生自矩陣中定義的責任類型：負責人 (R)、責任 (A)、諮詢 (C) 和知情 (I)。支援 (S) 類型為選用。如果您包含支援，則矩陣稱為 RASCI 矩陣，如果您排除它，則稱為 RACI 矩陣。

回應性控制

一種安全控制，旨在驅動不良事件或偏離安全基準的補救措施。如需詳細資訊，請參閱在 AWS上實作安全控制中的回應性控制。

保留

請參閱 7 個 R。

淘汰

請參閱 7 Rs。

檢索增強生成 (RAG)

一種生成式 AI 技術，其中 LLM 會在產生回應之前參考訓練資料來源以外的授權資料來源。例如，RAG 模型可能會對組織的知識庫或自訂資料執行語意搜尋。如需詳細資訊，請參閱什麼是 RAG。

輪換

定期更新秘密的程序，讓攻擊者更難存取登入資料。

資料列和資料欄存取控制 (RCAC)

使用已定義存取規則的基本、彈性 SQL 表達式。RCAC 包含資料列許可和資料欄遮罩。

RPO

請參閱復原點目標。

RTO

請參閱復原時間目標。

執行手冊

執行特定任務所需的一組手動或自動程序。這些通常是為了簡化重複性操作或錯誤率較高的程序而建置。

S

SAML 2.0

許多身分提供者 (IdP) 使用的開放標準。此功能會啟用聯合單一登入 (SSO)，讓使用者可以登入 AWS Management Console 或呼叫 AWS API 操作，而不必為您組織中的每個人在 IAM 中建立使用者。如需有關以 SAML 2.0 為基礎的聯合詳細資訊，請參閱 IAM 文件中的關於以 SAML 2.0 為基礎的聯合。

SCADA

請參閱監督控制和資料擷取。

SCP

請參閱服務控制政策。

秘密

您以加密形式存放的 AWS Secrets Manager機密或限制資訊，例如密碼或使用者登入資料。它由秘密值及其中繼資料組成。秘密值可以是二進位、單一字串或多個字串。如需詳細資訊，請參閱 Secrets Manager 文件中的 Secrets Manager 秘密中的什麼內容？。

依設計的安全性

透過整個開發程序將安全性納入考量的系統工程方法。

安全控制

一種技術或管理防護機制，它可預防、偵測或降低威脅行為者利用安全漏洞的能力。安全控制有四種主要類型：預防性、偵測性、回應性和主動性。

安全強化

減少受攻擊面以使其更能抵抗攻擊的過程。這可能包括一些動作，例如移除不再需要的資源、實作授予最低權限的安全最佳實務、或停用組態檔案中不必要的功能。

安全資訊與事件管理 (SIEM) 系統

結合安全資訊管理 (SIM) 和安全事件管理 (SEM) 系統的工具與服務。SIEM 系統會收集、監控和分析來自伺服器、網路、裝置和其他來源的資料，以偵測威脅和安全漏洞，並產生提醒。

安全回應自動化

預先定義和程式設計的動作，旨在自動回應或修復安全事件。這些自動化可做為偵測或回應式安全控制，協助您實作 AWS 安全最佳實務。自動化回應動作的範例包括修改 VPC 安全群組、修補 Amazon EC2 執行個體或輪換登入資料。

伺服器端加密

由接收資料的 AWS 服務 在其目的地加密資料。

服務控制政策 (SCP)

為 AWS Organizations中的組織的所有帳戶提供集中控制許可的政策。SCP 會定義防護機制或設定管理員可委派給使用者或角色的動作限制。您可以使用 SCP 作為允許清單或拒絕清單，以指定允許或禁止哪些服務或動作。如需詳細資訊，請參閱 AWS Organizations 文件中的服務控制政策。

服務端點

的進入點 URL AWS 服務。您可以使用端點，透過程式設計方式連接至目標服務。如需詳細資訊，請參閱 AWS 一般參考 中的 AWS 服務 端點。

服務水準協議 (SLA)

一份協議，闡明 IT 團隊承諾向客戶提供的服務，例如服務正常執行時間和效能。

服務層級指標 (SLI)

服務效能方面的測量，例如其錯誤率、可用性或輸送量。

服務層級目標 (SLO)

代表服務運作狀態的目標指標，由服務層級指標測量。

共同責任模式

描述您與 共同 AWS 承擔雲端安全與合規責任的模型。 AWS 負責雲端的安全，而 負責雲端的安全。如需詳細資訊，請參閱共同責任模式。

SIEM

請參閱安全資訊和事件管理系統。

單一故障點 (SPOF)

應用程式的單一關鍵元件故障，可能會中斷系統。

SLA

請參閱服務層級協議。

SLI

請參閱服務層級指標。

SLO

請參閱服務層級目標。

先拆分後播種模型

擴展和加速現代化專案的模式。定義新功能和產品版本時，核心團隊會進行拆分以建立新的產品團隊。這有助於擴展組織的能力和服務，提高開發人員生產力，並支援快速創新。如需詳細資訊，請參閱 中的階段式應用程式現代化方法 AWS 雲端。

SPOF

請參閱單一故障點。

星狀結構描述

使用一個大型事實資料表來存放交易或測量資料的資料庫組織結構，並使用一或多個較小的維度資料表來存放資料屬性。此結構旨在用於資料倉儲或商業智慧用途。

Strangler Fig 模式

一種現代化單一系統的方法，它會逐步重寫和取代系統功能，直到舊式系統停止使用為止。此模式源自無花果藤，它長成一棵馴化樹並最終戰勝且取代了其宿主。該模式由 Martin Fowler 引入，作為重寫單一系統時管理風險的方式。如需有關如何套用此模式的範例，請參閱使用容器和 Amazon API Gateway 逐步現代化舊版 Microsoft ASP.NET (ASMX) Web 服務。

子網

您 VPC 中的 IP 地址範圍。子網必須位於單一可用區域。

監控控制和資料擷取 (SCADA)

在製造中，使用硬體和軟體來監控實體資產和生產操作的系統。

對稱加密

使用相同金鑰來加密及解密資料的加密演算法。

合成測試

以模擬使用者互動的方式測試系統，以偵測潛在問題或監控效能。您可以使用 Amazon CloudWatch Synthetics 來建立這些測試。

系統提示

一種向 LLM 提供內容、指示或指導方針以指示其行為的技術。系統提示有助於設定內容，並建立與使用者互動的規則。

T

標籤

做為中繼資料的鍵/值對，用於組織您的 AWS 資源。標籤可協助您管理、識別、組織、搜尋及篩選資源。如需詳細資訊，請參閱標記您的 AWS 資源。

目標變數

您嘗試在受監督的 ML 中預測的值。這也被稱為結果變數。例如，在製造設定中，目標變數可能是產品瑕疵。

任務清單

用於透過執行手冊追蹤進度的工具。任務清單包含執行手冊的概觀以及要完成的一般任務清單。對於每個一般任務，它包括所需的預估時間量、擁有者和進度。

測試環境

請參閱 環境。

訓練

為 ML 模型提供資料以供學習。訓練資料必須包含正確答案。學習演算法會在訓練資料中尋找將輸入資料屬性映射至目標的模式 (您想要預測的答案)。它會輸出擷取這些模式的 ML 模型。可以使用 ML 模型，來預測您不知道的目標新資料。

傳輸閘道

可以用於互連 VPC 和內部部署網路的網路傳輸中樞。如需詳細資訊，請參閱 AWS Transit Gateway 文件中的什麼是傳輸閘道。

主幹型工作流程

這是一種方法，開發人員可在功能分支中本地建置和測試功能，然後將這些變更合併到主要分支中。然後，主要分支會依序建置到開發環境、生產前環境和生產環境中。

受信任的存取權

將許可授予您指定的服務，以代表您在組織中 AWS Organizations 及其帳戶中執行任務。受信任的服務會在需要該角色時，在每個帳戶中建立服務連結角色，以便為您執行管理工作。如需詳細資訊，請參閱 文件中的 AWS Organizations 搭配使用 AWS Organizations 與其他 AWS 服務。

調校

變更訓練程序的各個層面，以提高 ML 模型的準確性。例如，可以透過產生標籤集、新增標籤、然後在不同的設定下多次重複這些步驟來訓練 ML 模型，以優化模型。

雙比薩團隊

兩個比薩就能吃飽的小型 DevOps 團隊。雙披薩團隊規模可確保軟體開發中的最佳協作。

U

不確定性

這是一個概念，指的是不精確、不完整或未知的資訊，其可能會破壞預測性 ML 模型的可靠性。有兩種類型的不確定性：認知不確定性是由有限的、不完整的資料引起的，而隨機不確定性是由資料中固有的噪聲和隨機性引起的。如需詳細資訊，請參閱量化深度學習系統的不確定性指南。

未區分的任務

也稱為繁重工作，這是建立和操作應用程式的必要工作，但不為最終使用者提供直接價值或提供競爭優勢。未區分任務的範例包括採購、維護和容量規劃。

較高的環境

請參閱 環境。

V

清空

一種資料庫維護操作，涉及增量更新後的清理工作，以回收儲存並提升效能。

版本控制

追蹤變更的程序和工具，例如儲存庫中原始程式碼的變更。

VPC 對等互連

兩個 VPC 之間的連線，可讓您使用私有 IP 地址路由流量。如需詳細資訊，請參閱 Amazon VPC 文件中的什麼是 VPC 對等互連。

漏洞

危害系統安全性的軟體或硬體瑕疵。

W

暖快取

包含經常存取的目前相關資料的緩衝快取。資料庫執行個體可以從緩衝快取讀取，這比從主記憶體或磁碟讀取更快。

暖資料

不常存取的資料。查詢這類資料時，通常可接受中等速度的查詢。

視窗函數

SQL 函數，對與目前記錄在某種程度上相關的資料列群組執行計算。視窗函數適用於處理任務，例如根據目前資料列的相對位置計算移動平均值或存取資料列的值。

工作負載

提供商業價值的資源和程式碼集合，例如面向客戶的應用程式或後端流程。

工作串流

遷移專案中負責一組特定任務的功能群組。每個工作串流都是獨立的，但支援專案中的其他工作串流。例如，組合工作串流負責排定應用程式、波次規劃和收集遷移中繼資料的優先順序。組合工作串流將這些資產交付至遷移工作串流，然後再遷移伺服器和應用程式。

WORM

請參閱寫入一次，多次讀取。

WQF

請參閱AWS 工作負載資格架構。

寫入一次，讀取許多 (WORM)

儲存模型，可一次性寫入資料，並防止刪除或修改資料。授權使用者可以視需要多次讀取資料，但無法變更資料。此資料儲存基礎設施被視為不可變。

Z

零時差入侵

利用零時差漏洞的攻擊，通常是惡意軟體。

零時差漏洞

生產系統中未緩解的缺陷或漏洞。威脅行為者可以使用這種類型的漏洞來攻擊系統。開發人員經常因為攻擊而意識到漏洞。

零鏡頭提示

提供 LLM 執行任務的指示，但沒有可協助引導任務的範例 (快照)。LLM 必須使用其預先訓練的知識來處理任務。零鏡頭提示的有效性取決於任務的複雜性和提示的品質。另請參閱少量擷取提示。

殭屍應用程式

CPU 和記憶體平均使用率低於 5% 的應用程式。在遷移專案中，通常會淘汰這些應用程式。

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。