Patrones, arquitecturas e implementaciones de diseño en la nube

Recomendaciones de AWS

Recomendaciones de AWS: Patrones, arquitecturas e implementaciones de diseño en la nube

Table of Contents

Patrones, arquitecturas e implementaciones de diseño en la nube

Anitha Deenadayalan, Amazon Web Services (AWS)

mayo de 2024 (historial de documentos)

En esta guía se proporcionan indicaciones para implementar los patrones de diseño de modernización más utilizados mediante el uso de servicios de AWS. Un número cada vez mayor de aplicaciones modernas se diseña mediante arquitecturas de microservicios para lograr la escalabilidad, mejorar la velocidad de lanzamiento, reducir el alcance del impacto de los cambios y reducir la regresión. Esto permite mejorar la productividad de los desarrolladores y aumentar la agilidad, mejorar la innovación y centrarse más en las necesidades empresariales. Las arquitecturas de microservicios también admiten el uso de la mejor tecnología para el servicio y la base de datos, y promueven el código políglota y la persistencia políglota.

Tradicionalmente, las aplicaciones monolíticas se ejecutan en un único proceso, utilizan un almacén de datos y se ejecutan en servidores que se escalan verticalmente. En comparación, las aplicaciones de microservicios modernas son detalladas, tienen dominios de errores independientes, se ejecutan como servicios en toda la red y pueden usar más de un almacén de datos, según el caso de uso. Los servicios se escalan horizontalmente y una sola transacción puede abarcar varias bases de datos. Al desarrollar aplicaciones mediante arquitecturas de microservicios, los equipos de desarrollo deben centrarse en la comunicación de red, la persistencia políglota, la escalabilidad horizontal, la coherencia final y la gestión de las transacciones en los almacenes de datos. Por lo tanto, los patrones de modernización son fundamentales para resolver los problemas más frecuentes en el desarrollo de aplicaciones modernas y ayudan a acelerar la entrega de software.

Esta guía proporciona una referencia técnica para los arquitectos de la nube, los líderes técnicos, los propietarios de aplicaciones y empresas y los desarrolladores que deseen elegir la arquitectura de nube adecuada para los patrones de diseño basándose en prácticas recomendadas bien diseñadas. Cada patrón descrito en esta guía aborda uno o más escenarios conocidos en las arquitecturas de microservicios. En la guía se analizan los problemas y las consideraciones asociados a cada patrón, se proporciona una implementación arquitectónica de alto nivel y se describe la implementación de AWS para el patrón. Se proporcionan ejemplos de GitHub de código abierto y enlaces a talleres cuando están disponibles.

En la guía se cubren los siguientes patrones:

- Capa anticorrupción
- Patrones de enrutamiento de API:
  - Enrutamiento por nombres de host
  - Enrutamiento por rutas
  - Enrutamiento de encabezados HTTP
- Interruptor
- Aprovisionamiento de eventos
- Arquitectura hexagonal
- Publicación/suscripción
- Reintento con retroceso
- Patrones saga:
  - Coreografía de la saga
  - Orquestación de la saga
- Dispersión y recopilación
- Higo estrangulador
- Bandeja de salida transaccional

Resultados empresariales específicos

Al utilizar los patrones descritos en esta guía para modernizar las aplicaciones, puede:

- Diseñar e implementar arquitecturas confiables, seguras y eficientes desde el punto de vista operativo que estén optimizadas en términos de costo y rendimiento.
- Reducir la duración del ciclo de los casos de uso que requieren estos patrones, de modo que pueda centrarse en los desafíos específicos de la organización.
- Acelerar el desarrollo mediante la estandarización de las implementaciones de patrones con los servicios de AWS.
- Ayudar a los desarrolladores a crear aplicaciones modernas sin heredar deudas técnicas.

Patrón de la capa anticorrupción

Intención

El patrón de capa anticorrupción (ACL) actúa como una capa de mediación que traduce la semántica del modelo del dominio de un sistema a otro. Traduce el modelo del contexto limitado ascendente (monolito) a un modelo que se adapta al contexto limitado descendente (microservicio) antes de consumir el contrato de comunicación que establece el equipo ascendente. Este patrón podría aplicarse cuando el contexto limitado descendente contiene un subdominio central o el modelo ascendente es un sistema heredado que no se puede modificar. También reduce el riesgo de transformación y la interrupción de la actividad empresarial, ya que evita que los autores de la llamada cambien cuando sus llamadas se tienen que redirigir de manera transparente al sistema de destino.

Motivación

Durante el proceso de migración, cuando una aplicación monolítica se migra a los microservicios, podría haber cambios en la semántica del modelo del dominio del servicio recién migrado. Cuando las características del monolito sean necesarias para llamar a estos microservicios, las llamadas deben enrutarse al servicio migrado sin necesidad de hacer cambios en los servicios de llamadas. El patrón ACL permite al monolito llamar a los microservicios de manera transparente, ya que actúa como un adaptador o una capa de fachada que traduce las llamadas a la semántica más reciente.

Aplicabilidad

Considere la posibilidad de utilizar este patrón cuando se suceda lo siguiente:

- La aplicación monolítica actual tiene que comunicarse con una función que se migró a un microservicio y el modelo y la semántica del dominio del servicio migrado difieren de la característica original.
- Dos sistemas tienen una semántica distinta y deben intercambiar datos, pero no resulta práctico modificar un sistema para que sea compatible con el otro.
- Se recomienda utilizar un enfoque rápido y simplificado para adaptar un sistema a otro con un impacto mínimo.

- La aplicación se comunica con un sistema externo.

Problemas y consideraciones

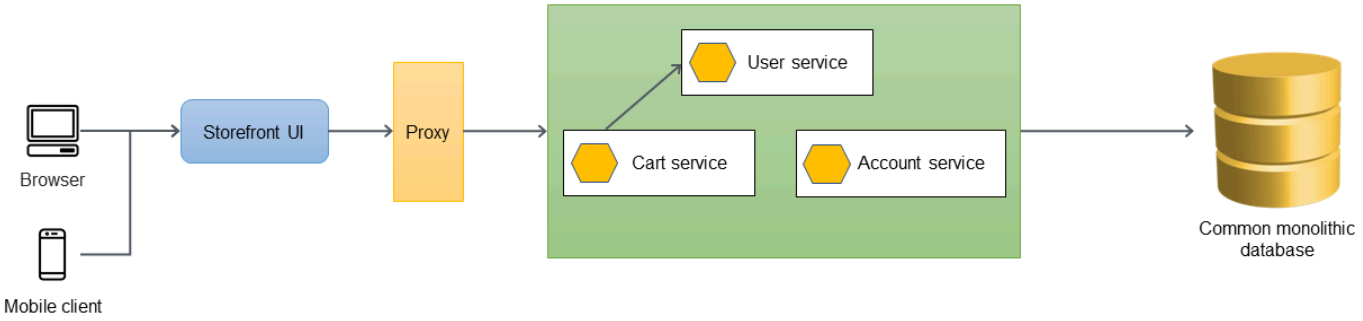
- Dependencias entre equipos. Cuando los servicios distintos de un sistema pertenecen a equipos distintos, la semántica nueva del modelo del dominio de los servicios migrados puede provocar cambios en los sistemas de llamadas. Sin embargo, es posible que los equipos no puedan hacer estos cambios de manera coordinada, porque es posible que tengan otras prioridades. La ACL desvincula a los destinatarios de las llamadas y traduce las llamadas para que coincidan con la semántica de los servicios nuevos, lo que evita que los autores de las llamadas tengan que hacer cambios en el sistema actual.
- Sobrecarga operativa. El funcionamiento y el mantenimiento del patrón de ACL requieren un esfuerzo adicional. Este trabajo incluye la integración de la ACL con herramientas de supervisión y alerta, el proceso de publicación y los procesos de integración y entrega continuas (CI/CD).
- Punto único de error. Los errores en la ACL pueden impedir la conexión con el servicio de destino y provocar problemas en las aplicaciones. Para mitigar este problema, debe incorporar funcionalidades de reintento y disyuntores. Consulte los patrones de reintento con retroceso y disyuntores para más información sobre estas opciones. La configuración de alertas y del registro adecuados mejorará el tiempo medio de la resolución (MTTR).
- Deuda técnica. Como parte de su estrategia de migración o modernización, considere si la ACL será una solución transitoria o provisional, o una solución a largo plazo. Si se trata de una solución provisional, debe registrar la ACL como deuda técnica y retirarla una vez que se hayan migrado todos los autores de llamadas dependientes que llamaron.
- Latencia. La capa adicional puede introducir latencia debido a la conversión de las solicitudes de una interfaz a otra. Le recomendamos definir y probar la tolerancia al rendimiento en las aplicaciones que son sensibles al tiempo de respuesta antes de implementar la ACL en los entornos de producción.
- Cuello de botella de escalado. En aplicaciones con cargas elevadas, en las que los servicios pueden escalar hasta alcanzar la carga máxima, la ACL puede convertirse en un cuello de botella y provocar problemas de escalabilidad. Si el servicio de destino se escala según la demanda, debe diseñar la ACL para que escale de manera adecuada.
- Implementación compartida o específica del servicio. Puede diseñar la ACL como un objeto compartido para convertir y redirigir las llamadas a varios servicios o clases de servicios específicos. Tenga en cuenta la latencia, el escalado y la tolerancia a los errores al determinar el tipo de implementación de la ACL.

Implementación

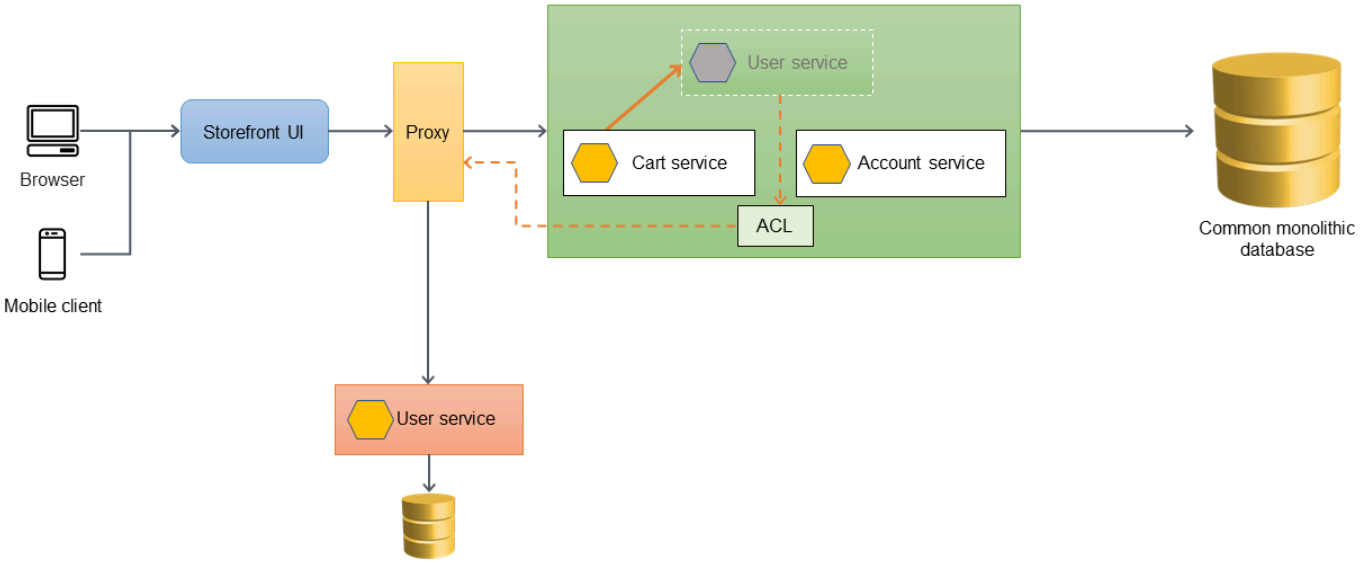
Puede implementar la ACL en la aplicación monolítica como una clase específica del servicio que se va a migrar o como un servicio independiente. La ACL debe retirarse una vez que todos los servicios dependientes se hayan migrado a la arquitectura de microservicios.

Arquitectura de alto nivel

En la arquitectura de ejemplo siguiente, una aplicación monolítica tiene tres servicios: servicio de usuario, servicio de carrito y servicio de cuentas. El servicio de carrito depende del servicio de usuario y la aplicación utiliza una base de datos relacional monolítica.

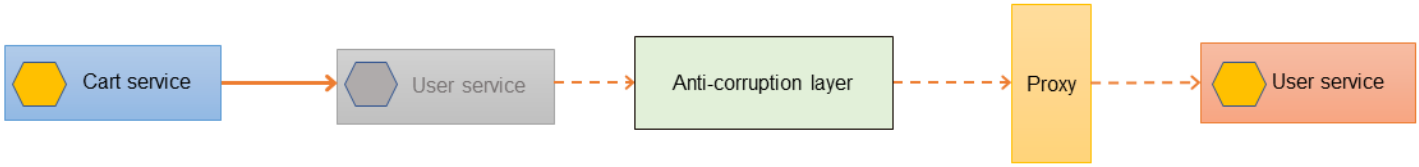


En la arquitectura siguiente, el servicio de usuario se migró a un microservicio nuevo. El servicio de carrito llama al servicio de usuario, pero la implementación deja de estar disponible en el monolito. También es probable que la interfaz del servicio recién migrado no coincida con la interfaz anterior, cuando estaba dentro de la aplicación monolítica.



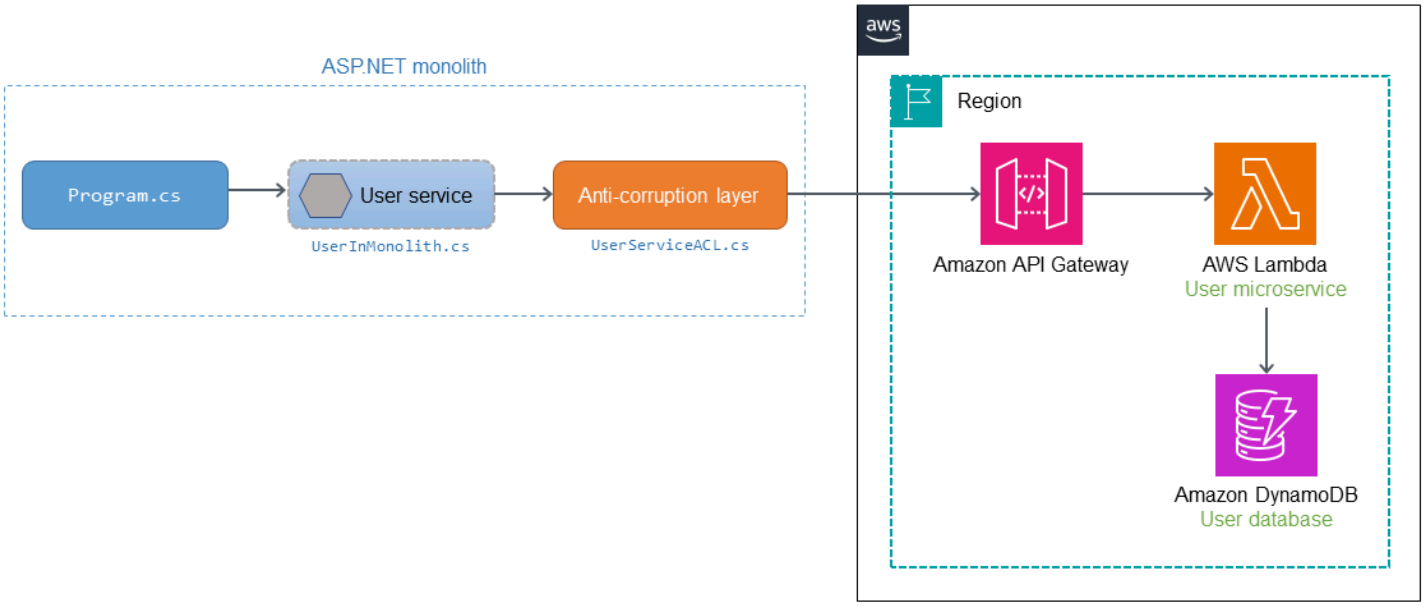
Si el servicio de carrito tiene que llamar de manera directa al servicio de usuario recién migrado, será necesario hacer cambios en el servicio de carrito y hacer pruebas exhaustivas de la aplicación monolítica. Esto puede aumentar el riesgo de transformación y la interrupción del negocio. La meta debe ser minimizar los cambios en la funcionalidad existente de la aplicación monolítica.

En este caso, le recomendamos introducir una ACL entre el servicio de usuario anterior y el servicio de usuario recién migrado. La ACL funciona como un adaptador o una fachada que convierte las llamadas en la interfaz más nueva. La ACL se puede implementar dentro de la aplicación monolítica como una clase (por ejemplo, `UserServiceFacade` o `UserServiceAdapter`) específica del servicio que se migró. La capa anticorrupción debe retirarse una vez que todos los servicios dependientes se hayan migrado a la arquitectura de microservicios.



Implementación mediante los servicios de AWS

En el diagrama siguiente se muestra cómo implementar este ejemplo de la ACL mediante el uso de los servicios de AWS.



El microservicio de usuario se migra de la aplicación monolítica de ASP.NET y se implementa como una función de AWS Lambda en AWS. Las llamadas a la función de Lambda se enrutan a través de

Amazon API Gateway. La ACL se implementa en el monolito para traducir la llamada y adaptarla a la semántica del microservicio del usuario.

Cuando `Program.cs` llama al servicio de usuario (`UserInMonolith.cs`) dentro del monolito, la llamada se enruta a la ACL (`UserServiceACL.cs`). La ACL traduce la llamada a la nueva semántica e interfaz, y llama al microservicio a través del punto de conexión de API Gateway. El autor de la llamada (`Program.cs`) no conoce la traducción y el enrutamiento que tienen lugar en el servicio de usuario y en la ACL. Como el autor de la llamada no está al tanto de los cambios en el código, se producen menos interrupciones en la actividad comercial y se reduce el riesgo de transformación.

Código de muestra

El fragmento de código siguiente proporciona los cambios en el servicio original y la implementación de `UserServiceACL.cs`. Cuando se recibe una solicitud, el servicio de usuario original llama a la ACL. La ACL convierte el objeto del origen para que coincida con la interfaz del servicio recién migrado, llama al servicio y devuelve la respuesta al autor de la llamada.

```
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;
    }
}
```

Repositorio GitHub

Para obtener una implementación completa de la arquitectura de ejemplo para este patrón, consulte el repositorio de GitHub en https://github.com/aws-samples/anti-corruption-layer-pattern.

Contenido relacionado

- Patrón de higo estrangulador
- Patrón de disyuntores
- Patrón de reintento con retroceso

Patrones de enrutamiento de API

En los entornos de desarrollo ágiles, los equipos autónomos (por ejemplo, escuadrones y tribus) poseen uno o más servicios que incluyen muchos microservicios. Los equipos exponen estos servicios como API para permitir a sus consumidores interactuar con su grupo de servicios y acciones.

Existen tres métodos principales para exponer las API HTTP a los principales consumidores mediante el uso de nombres de host y rutas:

Método	Descripción	Ejemplo
Enrutamiento por nombres de host	Se expone cada servicio como un nombre de host.	`billing.api.example.com`
Enrutamiento por rutas	Se expone cada servicio como una ruta.	`api.example.com/billing`
Enrutamiento basado en encabezados	Se expone cada servicio como un encabezado HTTP.	`x-example-action: something`

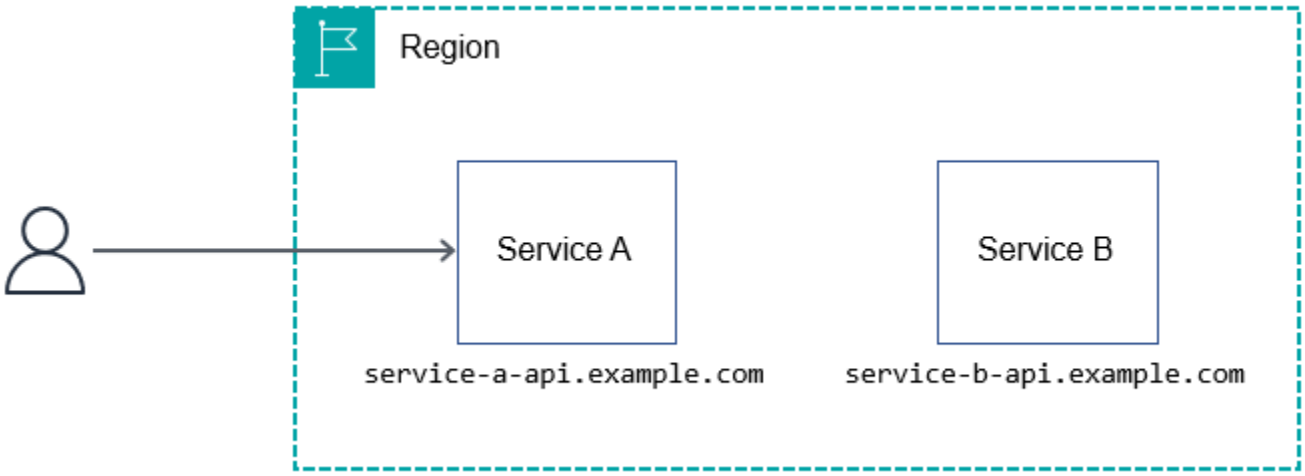
En esta sección se describen los casos de uso típicos de estos tres métodos de enrutamiento y sus ventajas y desventajas para ayudarle a decidir qué método se adapta mejor a sus requisitos y estructura organizativa.

Patrón de enrutamiento por nombres de host

El enrutamiento por nombre de host es un mecanismo para aislar los servicios de API al asignar a cada API su propio nombre de host; por ejemplo, `service-a.api.example.com` o `service-a.example.com`.

Caso de uso típico

El enrutamiento mediante nombres de host reduce la fricción en los lanzamientos, ya que los equipos de servicio no comparten nada. Los equipos son responsables de administrar todo, desde las entradas de DNS hasta las operaciones de servicio en producción.



Ventajas

El enrutamiento por nombres de host es, con diferencia, el método más sencillo y escalable para el enrutamiento de API HTTP. Puede utilizar cualquier servicio de AWS relevante para crear una arquitectura que siga este método: puede crear una arquitectura con Amazon API Gateway, AWS AppSync, equilibradores de carga de aplicación y Amazon Elastic Compute Cloud (Amazon EC2), o cualquier otro servicio compatible con HTTP.

Los equipos pueden usar el enrutamiento por nombres de host para ser plenamente propietarios de su subdominio. También facilita el aislamiento, las pruebas y la orquestación de las implementaciones para Regiones de AWS o versiones específicas; por ejemplo, `region.service-a.api.example.com` o `dev.region.service-a.api.example.com`.

Desventajas

Cuando se utiliza el enrutamiento por nombres de host, los consumidores tienen que recordar diferentes nombres de host para interactuar con cada API que se exponga. Puede mitigar este problema proporcionando un SDK cliente. Sin embargo, los SDK cliente presentan sus propios desafíos. Por ejemplo, deben admitir actualizaciones continuas, varios idiomas, control de versiones, comunicación de cambios importantes causados por problemas de seguridad o la corrección de errores, la documentación, etc.

Al utilizar el enrutamiento de nombres de host, también es necesario registrar el subdominio o el dominio cada vez que se crea un servicio nuevo.

Patrón de enrutamiento por rutas

El enrutamiento por rutas es el mecanismo que consiste en agrupar varias o todas las API bajo el mismo nombre de host y utilizar un URI de solicitud para aislar los servicios; por ejemplo, `api.example.com/service-a` o `api.example.com/service-b`.

Caso de uso típico

La mayoría de los equipos optan por este método porque quieren una arquitectura sencilla: el desarrollador solo tiene que recordar una URL, como `api.example.com`, para poder interactuar con la API HTTP. La documentación de la API suele ser más fácil de digerir porque suele estar reunida en lugar de estar dividida en diferentes portales o archivos PDF.

El enrutamiento basado en rutas se considera un mecanismo sencillo para compartir una API HTTP. Sin embargo, implica una sobrecarga operativa, como la configuración, la autorización, las integraciones y la latencia adicional debido a los múltiples saltos. También requiere procesos maduros de administración de cambios para garantizar que una mala configuración no interrumpa todos los servicios.

En AWS, hay varias formas de compartir una API y dirigirla de manera efectiva al servicio correcto. En las siguientes secciones se analizan tres enfoques: proxy inverso del servicio HTTP, API Gateway y Amazon CloudFront. Ninguno de los enfoques sugeridos para unificar los servicios de API se basa en los servicios descendentes que se ejecutan en AWS. Los servicios podrían ejecutarse en cualquier lugar sin problemas o con cualquier tecnología, siempre que sean compatibles con HTTP.

Proxy inverso del servicio HTTP

Puede utilizar un servidor HTTP como NGINX para crear configuraciones de enrutamiento dinámico. En una arquitectura de Kubernetes, también puede crear una regla de entrada que coincida con una ruta a un servicio. (Esta guía no cubre la entrada de Kubernetes; consulte la documentación de Kubernetes para obtener más información).

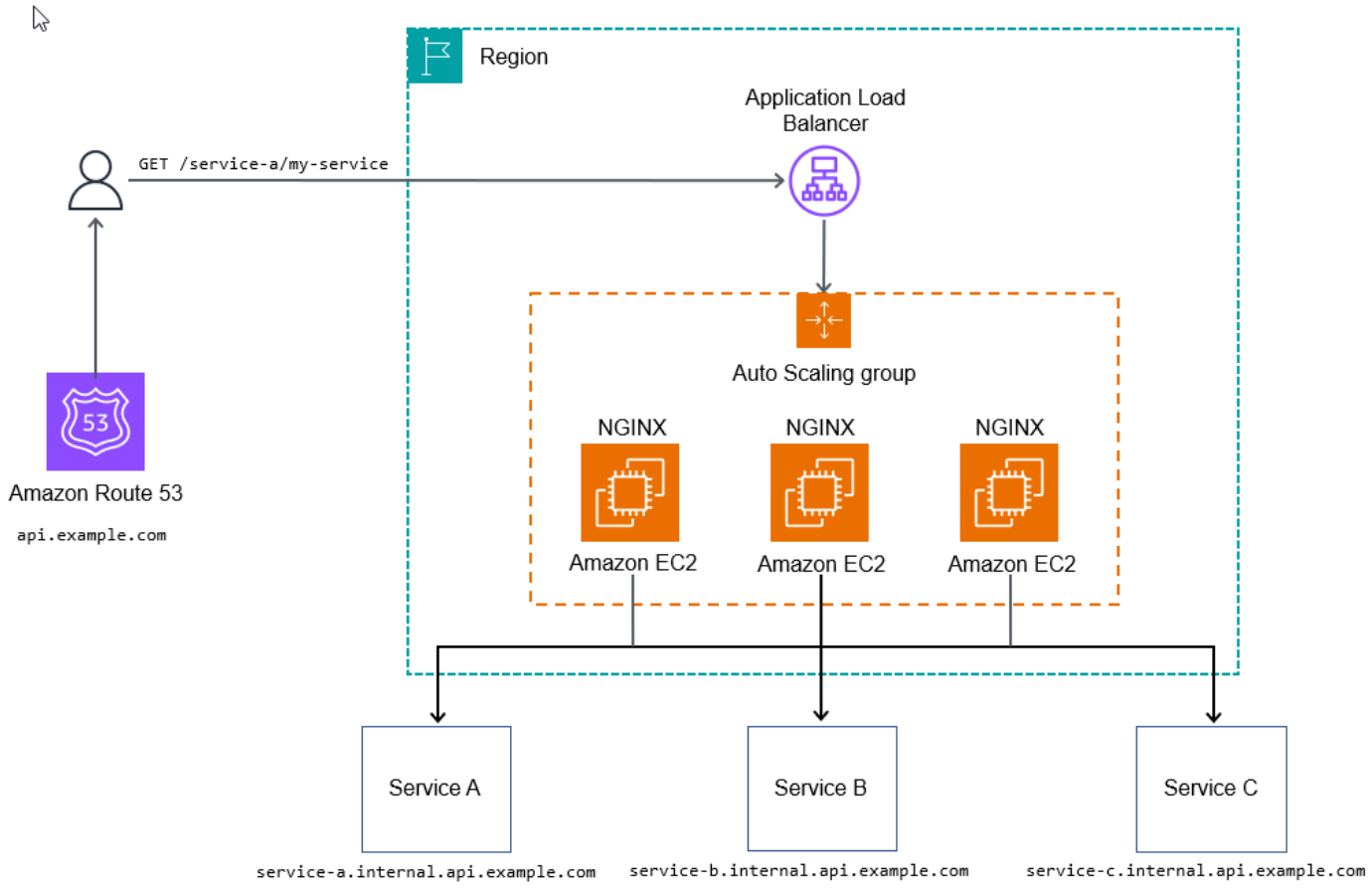
La siguiente configuración para NGINX asigna dinámicamente una solicitud HTTP de `api.example.com/my-service/` a `my-service.internal.api.example.com`.

```
server {
    listen  80;

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

```
    }
}
```

El siguiente diagrama ilustra el método de proxy inverso del servicio HTTP.



Este enfoque puede ser suficiente para algunos casos de uso en los que no se utilizan configuraciones adicionales para empezar a procesar las solicitudes, lo que permite que la API descendente recopile métricas y registros.

Para prepararse para la producción operativa, querrá poder agregar observabilidad a todos los niveles de su pila, añadir configuraciones adicionales o agregar scripts para personalizar el punto de entrada de la API y permitir características más avanzadas, como la limitación de velocidad o los tokens de uso.

Ventajas

El objetivo final del método de proxy inverso del servicio HTTP es crear un enfoque escalable y administrable para unificar las API en un solo dominio, de forma que resulte coherente para

cualquier consumidor de API. Este enfoque también permite a sus equipos de servicio implementar y administrar sus propias API, con una sobrecarga mínima después de la implementación. Los servicios administrados de AWS de rastreo, como AWS X-Ray o AWS WAF, siguen siendo aplicables en este caso.

Desventajas

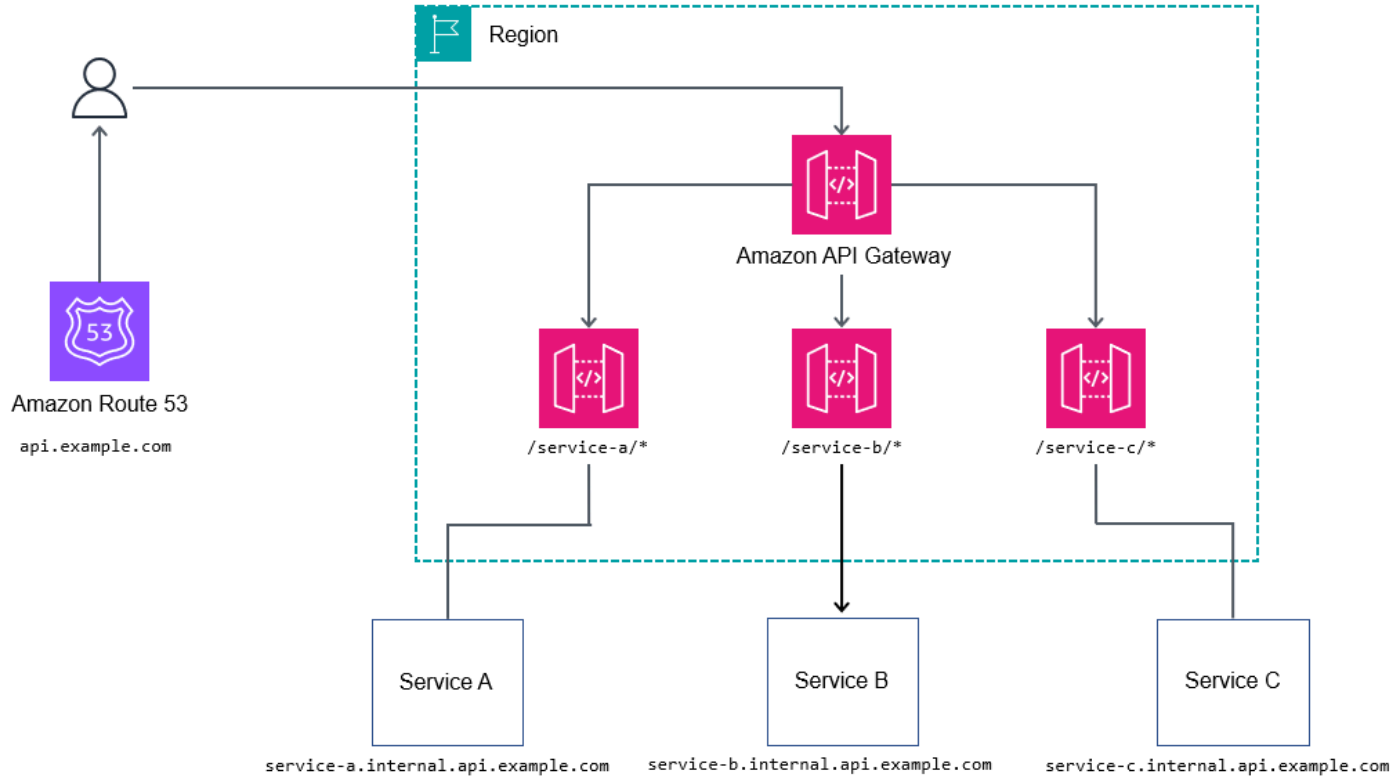
La principal desventaja de este enfoque son las exhaustivas pruebas y la administración de los componentes de la infraestructura que se requieren, aunque esto podría no ser un problema si se cuenta con equipos de ingeniería de confiabilidad del sitio (SRE).

Este método supone un punto de inflexión en cuanto a los costos. En volúmenes bajos o medianos, es más caro que algunos de los otros métodos descritos en esta guía. En volúmenes altos, es muy rentable (alrededor de 100 000 transacciones por segundo o más).

API Gateway

El servicio Amazon API Gateway (API de REST y API HTTP) puede enrutar el tráfico de forma similar al método de proxy inverso del servicio HTTP. El uso de una puerta de enlace API en modo proxy HTTP proporciona una forma sencilla de agrupar muchos servicios en un punto de entrada al subdominio de nivel superior `api.example.com` y, a continuación, enviar las solicitudes por proxy al servicio anidado; por ejemplo, `billing.internal.api.example.com`.

Probablemente no quiera ser demasiado detallado y asignar todas las rutas de todos los servicios en la puerta de enlace de API principal o raíz. En su lugar, opte por rutas comodín, por ejemplo, `/billing/*`, para reenviar las solicitudes al servicio de facturación. Al no asignar todas las rutas de la puerta de enlace de la API raíz o principal, se obtiene una mayor flexibilidad con respecto a las API, ya que no es necesario actualizar la puerta de enlace de la API raíz con cada cambio de API.



Ventajas

Para controlar los flujos de trabajo más complejos, como el cambio de los atributos de las solicitudes, las API de REST exponen Apache Velocity Template Language (VTL) para poder modificar la solicitud y la respuesta. Las API de REST pueden ofrecer beneficios adicionales como los siguientes:

- Autenticación N/Z con AWS Identity and Access Management (IAM), Amazon Cognito o autorizadores de AWS Lambda
- AWS X-Ray para el rastreo
- Cómo integrar con AWS WAF
- Limitación de velocidad básica
- Tokens de uso para agrupar en buckets a los consumidores en diferentes niveles (consulte Limitar las solicitudes de la API para mejorar el rendimiento en la documentación de API Gateway)

Desventajas

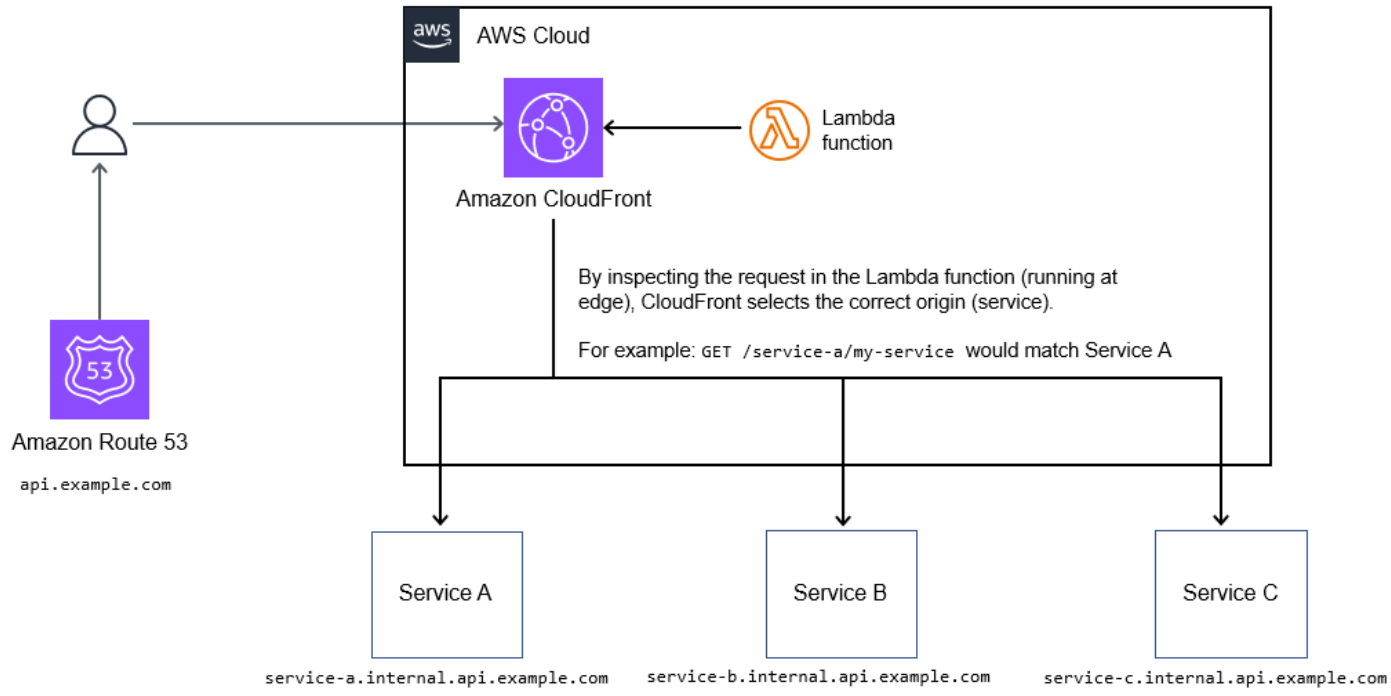
Con volúmenes altos, el costo puede ser un problema para algunos usuarios.

CloudFront

Puede utilizar la característica de selección dinámica de orígenes de Amazon CloudFront para seleccionar condicionalmente un origen (un servicio) para reenviar la solicitud. Puede utilizar esta característica para enrutar varios servicios a través de un único nombre de host, por ejemplo, `api.example.com`.

Caso de uso típico

La lógica de enrutamiento forma parte del código de la función de Lambda@Edge, por lo que admite mecanismos de enrutamiento altamente personalizables, como las pruebas A/B, las versiones de valor controlado, la señalización de características y la reescritura de rutas. Esto se explica en el siguiente diagrama.



Ventajas

Si necesita almacenar en caché las respuestas de la API, este método es una buena forma de unificar un conjunto de servicios en un único punto de conexión. Es un método rentable para unificar las colecciones de API.

Además, CloudFront admite el cifrado a nivel de campo, así como la integración con AWS WAF para la limitación de velocidad básica y las ACL básicas.

Desventajas

Este método admite un máximo de 250 orígenes (servicios) que se pueden unificar. Este límite es suficiente para la mayoría de las implementaciones, pero puede provocar problemas con una gran cantidad de API a medida que amplíe su cartera de servicios.

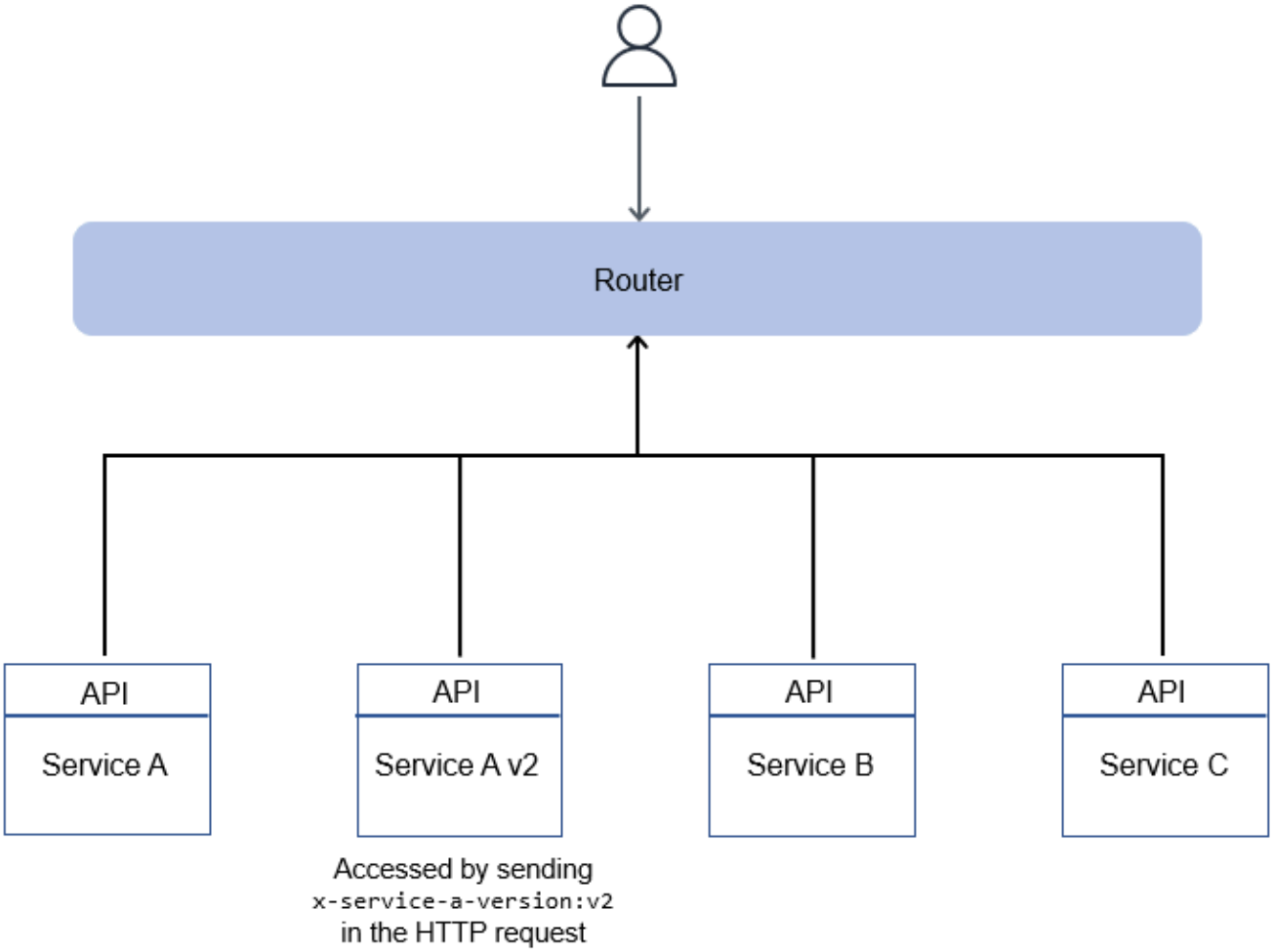
La actualización de las funciones de Lambda@Edge tarda unos minutos actualmente. CloudFront también tarda hasta 30 minutos en completar la propagación de los cambios a todos los puntos de presencia. En última instancia, esto bloquea las actualizaciones posteriores hasta que se completen.

Patrón de enrutamiento de encabezados HTTP

El enrutamiento basado en encabezados le permite dirigirse al servicio correcto para cada solicitud especificando un encabezado HTTP en la solicitud HTTP. Por ejemplo, enviar el encabezado `x-service-a-action: get-thing` le permitiría ir a `get thing` desde `Service A`. La ruta de la solicitud sigue siendo importante, ya que ofrece orientación sobre el recurso en el que está intentando trabajar.

Además de usar el enrutamiento de encabezados HTTP para las acciones, puede usarlo como un mecanismo para enrutar versiones, habilitar indicadores de características, pruebas A/B o necesidades similares. En realidad, es probable que utilice el enrutamiento de encabezados con uno de los otros métodos de enrutamiento para crear API sólidas.

La arquitectura del enrutamiento de encabezados HTTP suele tener una capa de enrutamiento delgada delante de los microservicios que se enruta al servicio correcto y devuelve una respuesta, como se ilustra en el siguiente diagrama. Esta capa de enrutamiento podría cubrir todos los servicios o solo algunos servicios para permitir una operación como el enrutamiento basado en versiones.



Ventajas

Los cambios de configuración requieren un esfuerzo mínimo y se pueden automatizar fácilmente. Este método también es flexible y admite formas creativas de exponer solo las operaciones específicas que se desearían realizar en un servicio.

Desventajas

Al igual que con el método de enrutamiento por nombres de host, el enrutamiento de encabezados HTTP supone que usted tiene el control total sobre el cliente y puede manipular encabezados HTTP personalizados. Los proxies, las redes de entrega de contenido (CDN) y los equilibradores de carga pueden limitar el tamaño del encabezado. Aunque es poco probable que esto sea motivo de preocupación, podría ser un problema según el número de encabezados y cookies que agregue.

Patrón de disyuntores

Intención

El patrón de disyuntores puede evitar que el servicio del autor de las llamadas vuelva a intentar hacer una llamada a otro servicio (destinatario) cuando la llamada haya provocado anteriormente tiempos de espera repetidos o erróneos. El patrón también se utiliza para detectar cuándo vuelve a funcionar el servicio del destinatario de las llamadas.

Motivación

Cuando varios microservicios colaboran para gestionar las solicitudes, es posible que uno o varios servicios dejen de estar disponibles o presenten una latencia alta. Cuando las aplicaciones complejas utilizan microservicios, la interrupción de un microservicio puede provocar que se produzcan errores en la aplicación. Los microservicios se comunican a través de llamadas a procedimientos remotos y pueden producirse errores transitorios en la conectividad de la red y provocar fallas. (Los errores transitorios se pueden gestionar mediante el patrón de reintento con retroceso). Durante la ejecución sincrónica, la acumulación de tiempos de espera o errores puede provocar una mala experiencia de usuario.

Sin embargo, en algunas situaciones, los errores pueden tardar más en resolverse. Por ejemplo, cuando el servicio del destinatario de la llamada está inactivo o cuando una contención en la base de datos provoca tiempos de espera. En esos casos, si el servicio que hace la llamada vuelve a intentar hacer las llamadas varias veces, estos reintentos pueden provocar una contención en la red y un consumo del conjunto de subprocesos de la base de datos. Además, si varios usuarios vuelven a intentar utilizar la aplicación en repetidas ocasiones, esto empeorará el problema y puede provocar que disminuya el rendimiento de toda la aplicación.

Michael Nygard popularizó el patrón de los disyuntores en su libro Release It (Nygard 2018). Este patrón de diseño puede evitar que el servicio del autor de la llamada vuelva a intentar hacer una llamada al servicio que anteriormente había provocado tiempos de espera o errores repetidos. También puede detectar cuándo vuelve a funcionar el servicio del autor de la llamada.

Los disyuntores funcionan como disyuntores eléctricos que interrumpen de manera automática la corriente cuando hay una anomalía en el circuito. Los disyuntores eléctricos cortan o interrumpen el flujo de corriente cuando hay una falla. Del mismo modo, el disyuntor está situado entre el autor de la

llamada y el servicio del destinatario de la llamada y se interrumpe si el destinatario de la llamada no está disponible.

Las falacias de la computación distribuida son un conjunto de afirmaciones de Peter Deutsch y otras personas de Sun Microsystems. Dicen que los programadores que se inician en el campo de las aplicaciones distribuidas hacen siempre suposiciones falsas. La fiabilidad de la red, las expectativas de latencia cero y las limitaciones de ancho de banda hacen que las aplicaciones de software se diseñen con una atención mínima a la gestión de los errores de red.

Durante una interrupción de la red, es posible que las aplicaciones esperen una respuesta de manera indefinida y consuman recursos de manera continua. Si no se vuelven a intentar las operaciones cuando la red está disponible, también se puede producir un deterioro de la aplicación. Si se agota el tiempo de espera de las llamadas de la API a una base de datos o a un servicio externo debido a problemas de red, las llamadas repetidas sin un disyuntor pueden afectar al costo y al rendimiento.

Aplicabilidad

Utilice este patrón cuando suceda lo siguiente:

- El servicio de del autor de la llamada hace una llamada en la que es muy probable que se produzcan errores.
- Si el servicio del destinatario de la llamada presenta una latencia alta (por ejemplo, cuando las conexiones a la base de datos son lentas), se agota el tiempo de espera del servicio del autor de la llamada.
- El servicio del autor de la llamada hace una llamada sincrónica, pero el servicio del destinatario de la llamada no está disponible o presenta una latencia alta.

Problemas y consideraciones

- Implementación independiente del servicio: para evitar la sobrecarga de código, le recomendamos implementar el objeto disyuntor de manera independiente de los microservicios y basada en la API.
- Cierre del circuito por parte del destinatario de la llamada: cuando el destinatario de la llamada se recupera de un problema o error de rendimiento, puede actualizar el estado del circuito a `CLOSED`. Se trata de una extensión del patrón de los disyuntores y se puede implementar si es necesario para el objetivo de tiempo de recuperación (RTO).

- Llamadas de varios subprocesos: el valor del tiempo de espera de vencimiento se define como el periodo que permanece desconectado el circuito antes de que se vuelvan a enrutar las llamadas para verificar la disponibilidad del servicio. Cuando se llama al servicio del destinatario de la llamada en varios subprocesos, la primera llamada erróneo define el valor del tiempo de espera de vencimiento. La implementación debe garantizar que las llamadas posteriores no prolonguen el tiempo de espera de vencimiento de manera indefinida.
- Fuerce la apertura o el cierre del circuito: los administradores del sistema deben poder abrir o cerrar un circuito. Para ello, se puede actualizar el valor del tiempo de espera de vencimiento en la tabla de la base de datos.
- Observabilidad: la aplicación debe tener un registro configurado para identificar las llamadas que presentan errores cuando está abierto el disyuntor.

Implementación

Arquitectura de alto nivel

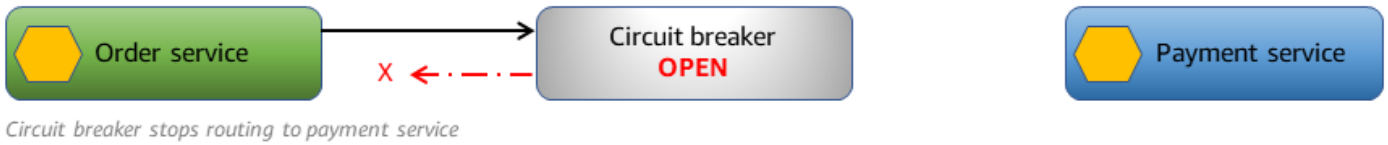
En el ejemplo siguiente, el autor de la llamada es el servicio de pedidos y el destinatario de la llamada es el servicio de pago.

Cuando no se produce ningún error, el servicio de pedidos dirige todas las llamadas al servicio de pago mediante el disyuntor, tal y como se muestra en el diagrama siguiente.

Si se agota el tiempo de espera del servicio de pago, el disyuntor puede detectar el tiempo de espera y hacer un seguimiento del error.

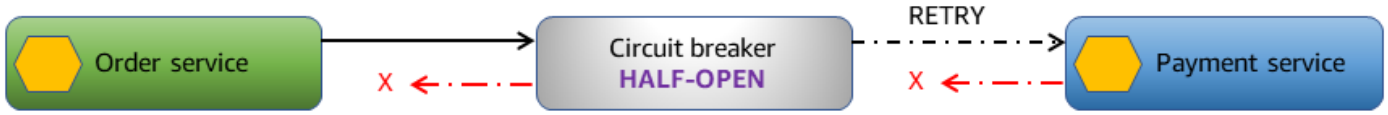
Circuit breaker with payment service failure

Si los tiempos de espera superan un umbral especificado, la aplicación abre el circuito. Cuando el circuito está abierto, el objeto del disyuntor no enruta las llamadas al servicio de pago. Se produce un error inmediato cuando el servicio de pedidos llama al servicio de pago.



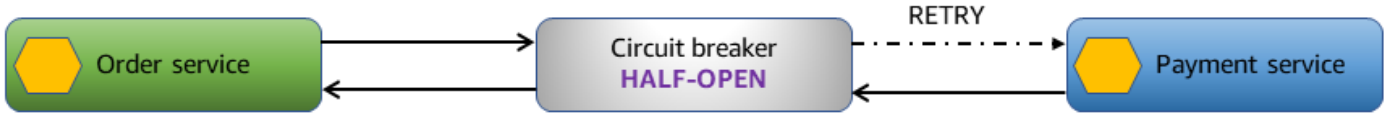
Circuit breaker stops routing to payment service

El objeto del disyuntor intenta verificar de manera periódica si las llamadas al servicio de pago se hicieron de manera correcta.



Circuit breaker periodically retries payment service

Cuando la llamada al servicio de pago se hace de manera correcta correctamente, el circuito se cierra y todas las demás llamadas se vuelven a dirigir al servicio de pago.



Circuit breaker with working payment service

Implementación mediante los servicios de AWS

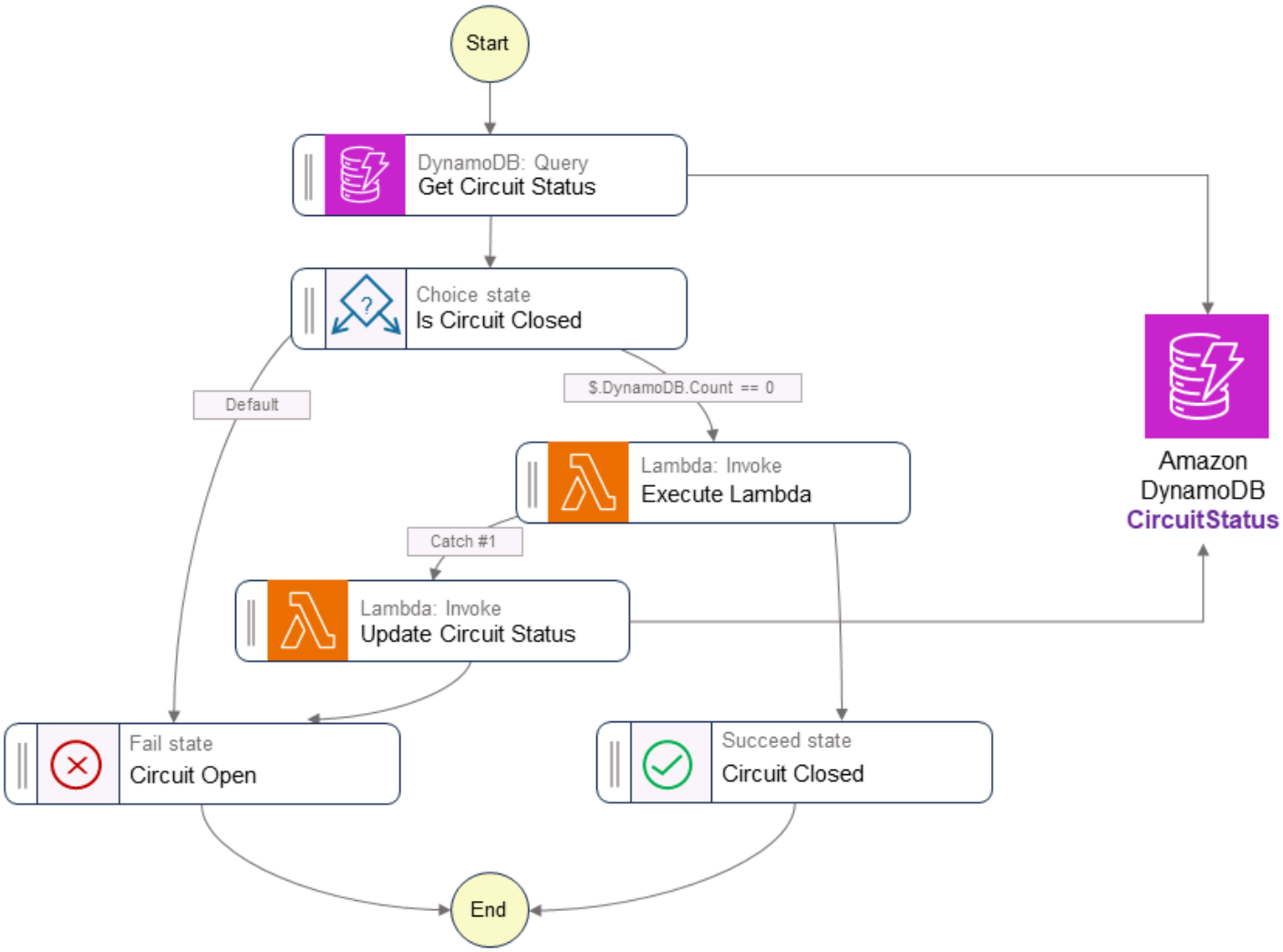
La solución de muestra utiliza flujos de trabajo rápidos en AWS Step Functions para implementar el patrón de disyuntores. La máquina de estados de Step Functions permite configurar las funcionalidades de reintento y el flujo de control basados en decisiones que son necesarios para la implementación del patrón.

La solución también utiliza una tabla de Amazon DynamoDB como almacén de datos para hacer un seguimiento del estado del circuito. Esto se puede sustituir por un almacén de datos en memoria, como Amazon ElastiCache (Redis OSS), para mejorar el rendimiento.

Cuando un servicio quiere llamar a otro servicio, inicia el flujo de trabajo con el nombre del servicio del destinatario de la llamada. El flujo de trabajo obtiene el estado del disyuntor de la tabla `CircuitStatus` de DynamoDB, que almacena los servicios actualmente degradados. Si `CircuitStatus` contiene un registro vigente del destinatario de la llamada, el circuito está abierto. El flujo de trabajo de Step Functions devuelve un error inmediato y se cierra con un estado `FAIL`.

Si la tabla `CircuitStatus` no contiene un registro del destinatario de la llamada o contiene un registro vencido, el servicio está operativo. El paso `ExecuteLambda` de la definición de la máquina

de estados llama a la función de Lambda que se envía a través de un valor de parámetro. Si la llamada es correcta, el flujo de trabajo de Step Functions finaliza con un estado `SUCCESS`.



Si se produce un error en la llamada del servicio o se agota el tiempo de espera, la aplicación vuelve a intentarlo con un retroceso exponencial durante un número definido de veces. Si se produce un error en la llamada del servicio después de los reintentos, el flujo de trabajo inserta un registro en la tabla `CircuitStatus` para el servicio con un valor `ExpiryTimeStamp` y el flujo de trabajo sale con un estado `FAIL`. Las llamadas posteriores al mismo servicio producen un error inmediato mientras el disyuntor esté abierto. El paso `Get Circuit Status` de la definición de la máquina de estados verifica la disponibilidad del servicio según el valor `ExpiryTimeStamp`. Los elementos vencidos se eliminan de la tabla `CircuitStatus` mediante la característica periodo de vida (TTL) de DynamoDB.

Código de muestra

El código siguiente utiliza la función de Lambda `GetCircuitStatus` para verifica el estado del disyuntor.

```
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 = "";
}
```

En el siguiente código se muestran las declaraciones de Amazon States Language en el flujo de trabajo de Step Functions.

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

Repositorio GitHub

Para obtener una implementación completa de la arquitectura de ejemplo para este patrón, consulte el repositorio de GitHub en https://github.com/aws-samples/circuit-breaker-netcore-blog.

Referencias de blogs

- Using the circuit breaker pattern with AWS Step Functions and Amazon DynamoDB

Contenido relacionado

- Patrón de higo estrangulador
- Patrón de reintento con retroceso
- Funcionalidades de disyuntores de AWS App Mesh

Patrón de aprovisionamiento de eventos

Intención

En las arquitecturas basadas en eventos, el patrón de aprovisionamiento de eventos almacena los eventos que provocan un cambio de estado en un almacén de datos. Esto ayuda a capturar y mantener un historial completo de los cambios de estado y promueve la auditabilidad, la trazabilidad y la capacidad de analizar los estados pasados.

Motivación

Varios microservicios pueden colaborar para gestionar las solicitudes y se comunican a través de eventos. Estos eventos pueden provocar un cambio de estado (datos). El almacenamiento de los objetos de eventos en el orden en que se producen proporciona información valiosa sobre el estado actual de la entidad de datos e información adicional sobre cómo llegó a ese estado.

Aplicabilidad

Utilice el patrón de aprovisionamiento de eventos cuando:

- Para el seguimiento, se requiere un historial inmutable de los eventos que se producen en una aplicación.
- Las proyecciones de datos políglotas se requieren a partir de una fuente única de información fiable (SSOT).
- Es necesaria una reconstrucción en un momento dado del estado de la aplicación.
- No es necesario almacenar el estado de la aplicación a largo plazo, pero es posible que desee reconstruirlo según sea necesario.
- Las cargas de trabajo tienen diferentes volúmenes de lectura y escritura. Por ejemplo, tiene cargas de trabajo de escritura intensiva que no requieren procesamiento en tiempo real.
- La captura de datos de cambios (CDC) es necesaria para analizar el rendimiento de las aplicaciones y otras métricas.
- Los datos de auditoría son necesarios para todos los eventos que ocurren en un sistema con fines de presentación de informes y cumplimiento.

- Para obtener escenarios hipotéticos, cambie (inserte, actualice o elimine) los eventos durante el proceso de reproducción para determinar el posible estado final.

Problemas y consideraciones

- Control de simultaneidad optimista: este patrón almacena todos los eventos que provocan un cambio de estado en el sistema. Varios usuarios o servicios pueden intentar actualizar el mismo dato al mismo tiempo, lo que provoca colisiones de eventos. Estas colisiones se producen cuando se crean y aplican eventos conflictivos al mismo tiempo, lo que da como resultado un estado final de los datos que no se corresponde con la realidad. Para solucionar este problema, puede implementar estrategias para detectar y resolver las colisiones de eventos. Por ejemplo, puede implementar un esquema de control de simultaneidad optimista incluyendo el control de versiones o agregando marcas de tiempo a los eventos para hacer un seguimiento del orden de las actualizaciones.
- Complejidad: la implementación del aprovisionamiento de eventos requiere un cambio de mentalidad, pasando de las operaciones tradicionales de CRUD a una mentalidad basada en eventos. El proceso de reproducción, que se utiliza para restaurar el sistema a su estado original, puede resultar complejo para garantizar la idempotencia de los datos. El almacenamiento de eventos, las copias de seguridad y las instantáneas también pueden añadir complejidad adicional.
- Coherencia de eventos: las proyecciones de datos derivadas de los eventos son coherentes finalmente debido a la latencia en la actualización de los datos mediante el patrón de división de responsabilidades por consultas de comandos (CQRS) o vistas materializadas. Cuando los consumidores procesan datos de un almacén de eventos y los editores envían nuevos datos, es posible que la proyección de datos o el objeto de la aplicación no representen el estado actual.
- Consultas: la recuperación de datos actuales o agregados de los registros de eventos puede ser más compleja y lenta en comparación con las bases de datos tradicionales, especialmente para consultas complejas y tareas de generación de informes. Para mitigar este problema, el aprovisionamiento de eventos se suele implementar con el patrón CQRS.
- Tamaño y costo del almacén de eventos: el almacén de eventos puede experimentar un crecimiento exponencial debido a la persistencia continua de los eventos, especialmente en sistemas con un alto rendimiento de eventos o periodos de retención prolongados. Por lo tanto, debe archivar periódicamente los datos de los eventos en un almacenamiento rentable para evitar que el almacén de eventos se agrande demasiado.
- Escalabilidad del almacén de eventos: el almacén de eventos debe gestionar de manera eficiente grandes volúmenes de operaciones de escritura y lectura. Escalar un almacén de eventos

puede resultar difícil, por lo que es importante contar con un almacén de datos que proporcione particiones.

- Eficiencia y optimización: elija o diseñe un almacén de eventos que gestione las operaciones de escritura y lectura de forma eficiente. El almacén de eventos debe optimizarse para el volumen de eventos y los patrones de consulta esperados para la aplicación. La implementación de mecanismos de indexación y consulta puede acelerar la recuperación de eventos al reconstruir el estado de la aplicación. También puede considerar la posibilidad de utilizar bibliotecas o bases de datos de almacenes de eventos especializadas que ofrezcan características de optimización de consultas.
- Instantáneas: debe realizar copias de seguridad de los registros de eventos a intervalos regulares con una activación en función del tiempo. Si se reproducen los eventos de la última copia de seguridad correcta de la que se tenga constancia, se recuperará el estado de la aplicación en un momento dado. El objetivo de punto de recuperación (RPO) es el tiempo máximo aceptable desde el último punto de recuperación de datos. El RPO determina qué se considera una pérdida de datos aceptable entre el último punto de recuperación y la interrupción del servicio. La frecuencia de las instantáneas diarias del almacén de datos y eventos debe basarse en el RPO de la aplicación.
- Sensibilidad temporal: los eventos se almacenan en el orden en que se producen. Por lo tanto, la fiabilidad de la red es un factor importante a tener en cuenta al implementar este patrón. Los problemas de latencia pueden provocar un estado incorrecto del sistema. Utilice las colas “primero en entrar, primero en salir” (FIFO, por sus siglas en inglés) con una entrega como máximo para llevar los eventos al almacén de eventos.
- Rendimiento de reproducción de eventos: reproducir un número considerable de eventos para reconstruir el estado actual de la aplicación puede llevar mucho tiempo. Se requieren esfuerzos de optimización para mejorar el rendimiento, especialmente cuando se reproducen eventos a partir de datos archivados.
- Actualizaciones externas del sistema: las aplicaciones que utilizan el patrón de aprovisionamiento de eventos pueden actualizar los almacenes de datos de sistemas externos y capturar estas actualizaciones como objetos de eventos. Durante la reproducción de los eventos, esto podría convertirse en un problema si el sistema externo no espera ninguna actualización. En esos casos, puede utilizar los indicadores de características para controlar las actualizaciones externas del sistema.
- Consultas al sistema externo: cuando las llamadas al sistema externo son sensibles a la fecha y hora de la llamada, los datos recibidos se pueden almacenar en almacenes de datos internos para utilizarlos durante las reproducciones.

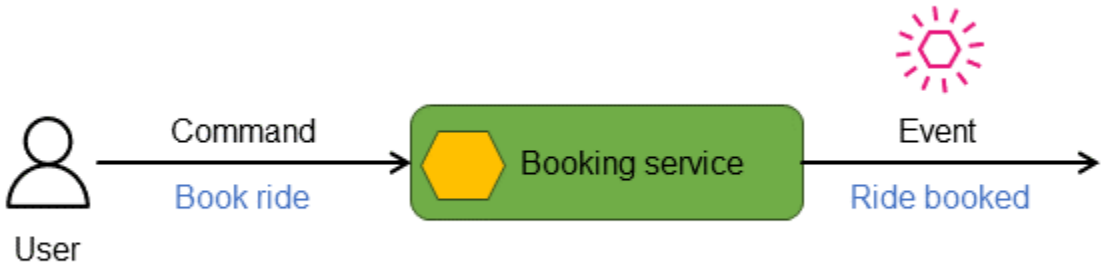
- Control de versiones de eventos: a medida que la aplicación evoluciona, la estructura de los eventos (esquema) puede cambiar. Es necesario implementar una estrategia de control de versiones para los eventos a fin de garantizar la compatibilidad con versiones anteriores y posteriores. Esto puede implicar la inclusión de un campo de versión en la carga útil del evento y la gestión de las diferentes versiones del evento de forma adecuada durante la reproducción.

Implementación

Arquitectura de alto nivel

Comandos y eventos

En las aplicaciones de microservicios distribuidas y basadas en eventos, los comandos representan las instrucciones o solicitudes enviadas a un servicio, normalmente con la intención de iniciar un cambio en su estado. El servicio procesa estos comandos y evalúa su validez y aplicabilidad en su estado actual. Si el comando se ejecuta correctamente, el servicio responde emitiendo un evento que indica la acción realizada y la información de estado relevante. Por ejemplo, en el siguiente diagrama, el servicio de reservas responde al comando Reservar viaje emitiendo el evento Viaje reservado.



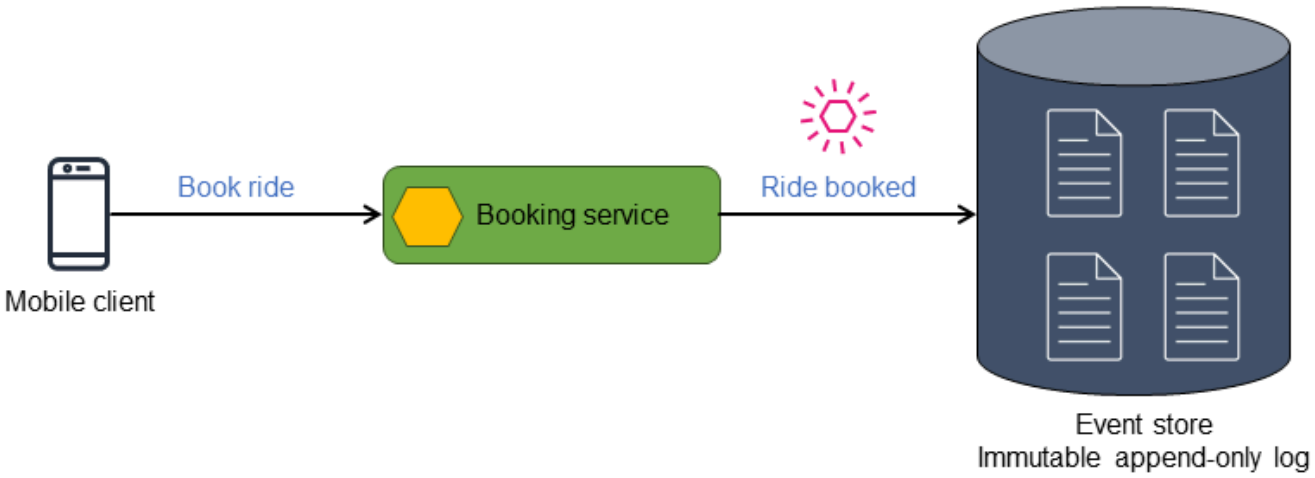
Almacenes de eventos

Los eventos se registran en un repositorio o almacén de datos inmutable, solo de anexos y ordenado cronológicamente, conocido como almacén de eventos. Cada cambio de estado se trata como un objeto de evento individual. Un objeto de entidad o un almacén de datos con un estado inicial conocido, su estado actual y cualquier vista en un momento dado se pueden reconstruir reproduciendo los eventos en el orden en que se produjeron.

El almacén de eventos actúa como un registro histórico de todas las acciones y cambios de estado, y sirve como una valiosa fuente única de información fiable. Puede utilizar el almacén de eventos para obtener el estado final y actualizado del sistema pasando los eventos por un procesador de reproducción, que los aplica para producir una representación precisa del estado más reciente del sistema. También puede utilizar el almacén de eventos para generar una perspectiva en un momento dado del estado mediante la reproducción de los eventos a través de un procesador de reproducción. En el patrón de aprovisionamiento de eventos, es posible que el estado actual no esté completamente representado por el objeto de evento más reciente. Puede obtener el estado actual de tres maneras:

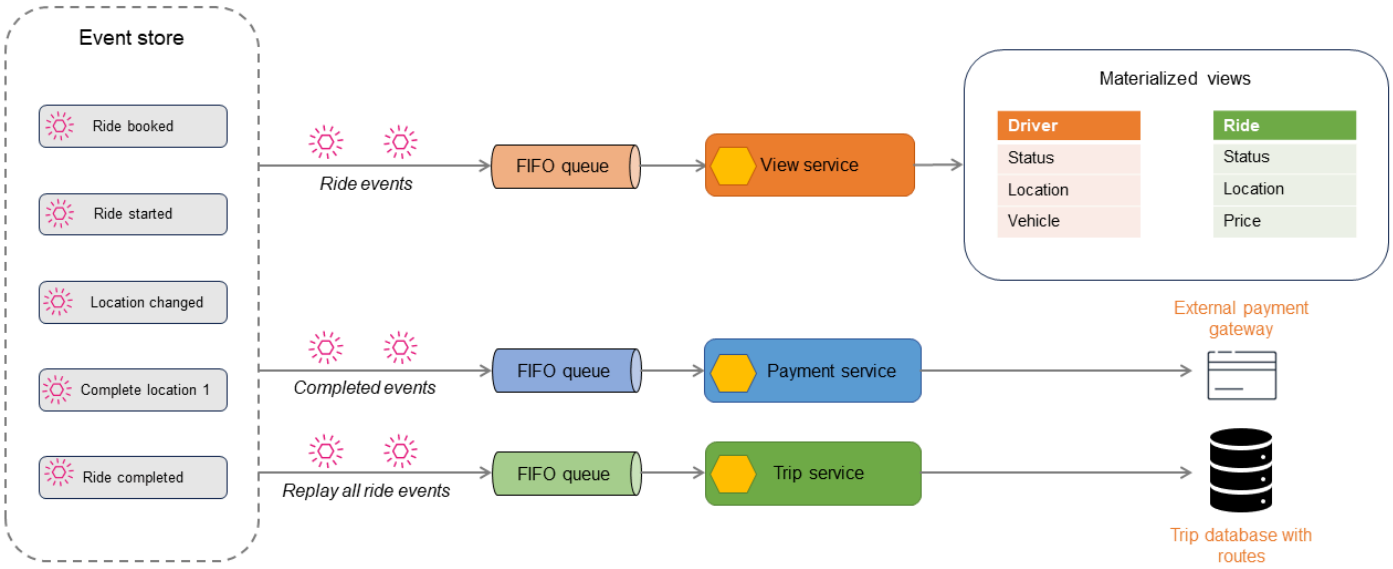
- Mediante la agregación de eventos relacionados. Los objetos de eventos relacionados se combinan para generar el estado actual para la consulta. Este enfoque se suele utilizar junto con el patrón CQRS, ya que los eventos se combinan y se escriben en el almacén de datos de solo lectura.
- Mediante el uso de vistas materializadas. Puede utilizar el aprovisionamiento de eventos con el patrón de vista materializada para calcular o resumir los datos del evento y obtener el estado actual de los datos relacionados.
- Mediante la reproducción de eventos. Los objetos de eventos se pueden reproducir para llevar a cabo acciones que generen el estado actual.

El siguiente diagrama muestra el evento `Ride booked` almacenado en un almacén de eventos.



El almacén de eventos publica los eventos que almacena, y los eventos se pueden filtrar y enrutar al procesador correspondiente para realizar acciones posteriores. Por ejemplo, los eventos se pueden enrutar a un procesador de vistas que resuma el estado y muestre una vista materializada. Los eventos se transforman al formato de datos del almacén de datos de destino. Esta arquitectura se puede ampliar para derivar diferentes tipos de almacenes de datos, lo que conduce a una persistencia políglota de los datos.

En el diagrama siguiente, se describen los eventos de una aplicación de reserva de viajes. Todos los eventos que se producen en la aplicación se almacenan en el almacén de eventos. A continuación, los eventos almacenados se filtran y se envían a diferentes consumidores.



Los eventos de los viajes se pueden utilizar para generar almacenes de datos de solo lectura mediante el CQRS o el patrón de vista materializada. Para obtener el estado actual del viaje, del conductor o de la reserva, consulte los almacenes de lectura. Algunos eventos, como `Location changed` o `Ride completed`, se publican para otro consumidor para el procesamiento de pagos. Cuando se completa el viaje, todos los eventos del viaje se reproducen para crear un historial del viaje con fines de auditoría o elaboración de informes.

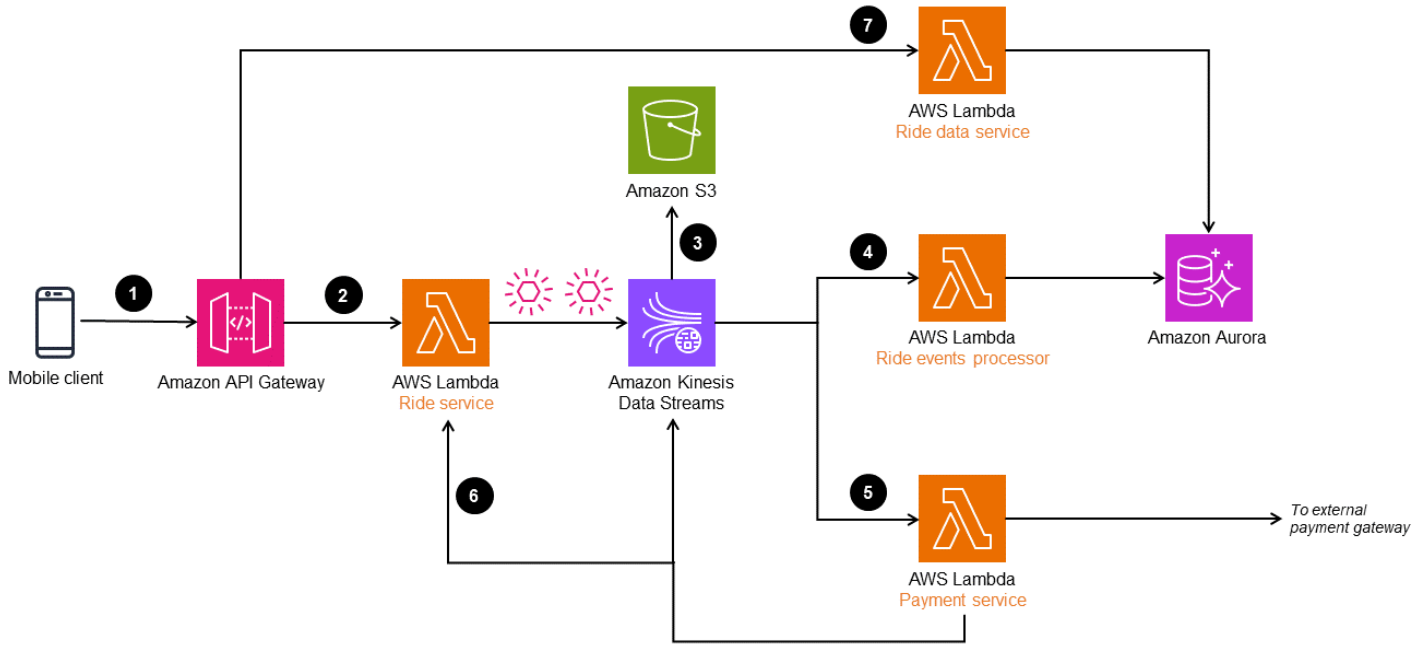
El patrón de aprovisionamiento de eventos se utiliza con frecuencia en aplicaciones que requieren una recuperación en un momento dado y también cuando los datos deben proyectarse en diferentes formatos utilizando una fuente única de información fiable. Ambas operaciones requieren un proceso de reproducción para ejecutar los eventos y obtener el estado final requerido. Es posible que el procesador de reproducción también requiera un punto de partida conocido, idealmente no desde el inicio de la aplicación, ya que no sería un proceso eficiente. Le recomendamos que tome

instantáneas periódicas del estado del sistema y aplique un número menor de eventos para obtener un estado actualizado.

Implementación mediante los servicios de AWS

En la siguiente arquitectura, Amazon Kinesis Data Streams se utiliza como almacén de eventos. Este servicio captura y administra los cambios en las aplicaciones como eventos y ofrece una solución de flujo de datos en tiempo real y de alto rendimiento. Para implementar el patrón de aprovisionamiento de eventos en AWS, también puede utilizar servicios como Amazon EventBridge y Amazon Managed Streaming para Apache Kafka (Amazon MSK) en función de las necesidades de su aplicación.

Para mejorar la durabilidad y habilitar la auditoría, puede archivar los eventos capturados por Kinesis Data Streams en Amazon Simple Storage Service (Amazon S3). Este enfoque de almacenamiento doble ayuda a retener los datos de eventos históricos de forma segura para futuros análisis y fines de cumplimiento.



El flujo de trabajo consta de los siguientes pasos:

1. La solicitud de reserva de un viaje se realiza a través de un cliente móvil a un punto de conexión de Amazon API Gateway.
2. El microservicio de viajes (función de Lambda `Ride service`) recibe la solicitud, transforma los objetos y los publica en Kinesis Data Streams.

3. Los datos de eventos de Kinesis Data Streams se almacenan en Amazon S3 con fines de cumplimiento e historial de auditoría.
4. La función de Lambda `Ride event processor` transforma y procesa los eventos y los almacena en una base de datos de Amazon Aurora para proporcionar una vista materializada de los datos del viaje.
5. Los eventos de viajes completados se filtran y se envían para su procesamiento a una puerta de enlace de pago externa. Cuando se haya completado el pago, se enviará otro evento a Kinesis Data Streams para actualizar la base de datos del viaje.
6. Cuando se completa el viaje, los eventos del viaje se reproducen en la función de Lambda `Ride service` para crear las rutas y el historial del viaje.
7. La información sobre los viajes se puede leer a través del `Ride data service`, que se lee en la base de datos de Aurora.

API Gateway también puede enviar el objeto de evento directamente a Kinesis Data Streams sin la función de Lambda `Ride service`. Sin embargo, en un sistema complejo, como un servicio de transporte privado, es posible que sea necesario procesar y enriquecer el objeto del evento antes de incorporarlo al flujo de datos. Por este motivo, la arquitectura tiene un `Ride service` que procesa el evento antes de enviarlo a Kinesis Data Streams.

Referencias de blogs

- Novedades de AWS Lambda: SQS FIFO como origen de eventos

Patrón de arquitectura hexagonal

Intención

En 2005, el Dr. Alistair Cockburn propuso el patrón de arquitectura hexagonal, también denominado patrón de puertos y adaptadores. Su objetivo es crear arquitecturas de acoplamiento débil en las que los componentes de las aplicaciones se puedan probar de manera independiente, sin depender de los almacenes de datos ni de las interfaces de usuario (UI). Este patrón ayuda a evitar el bloqueo tecnológico de los almacenes de datos y las interfaces de usuario. Esto facilita cambiar el conjunto de tecnologías a lo largo del tiempo, con un impacto limitado o nulo en la lógica empresarial. En esta arquitectura de acoplamiento débil, la aplicación se comunica con los componentes externos a través de interfaces denominadas puertos y utiliza adaptadores para traducir los intercambios técnicos con estos componentes.

Motivación

El patrón de arquitectura hexagonal se utiliza para aislar la lógica empresarial (lógica de dominio) del código de infraestructura relacionado, como el código para acceder a una base de datos o a las API externas. Este patrón resulta útil para crear una lógica empresarial y un código de infraestructura acoplados de manera débil para las funciones AWS Lambda que requieren la integración con servicios externos. En las arquitecturas tradicionales, una práctica habitual es integrar la lógica empresarial en la capa de bases de datos como procedimientos almacenados y en la interfaz de usuario. Esta práctica, junto con el uso de constructos específicos de la interfaz de usuario en la lógica empresarial, da como resultado arquitecturas de acoplamiento ajustado que provocan cuellos de botella en las migraciones de bases de datos y en los esfuerzos de modernización de la experiencia de usuario (UX). El patrón de arquitectura hexagonal le permite diseñar los sistemas y aplicaciones según la finalidad propósito y no según la tecnología. Esta estrategia da como resultado componentes de aplicaciones que se pueden intercambiar de manera fácil, como bases de datos, experiencia de usuario y componentes de servicio.

Aplicabilidad

Utilice el patrón de arquitectura hexagonal cuando suceda lo siguiente:

- Desea desacoplar la arquitectura de la aplicación para crear componentes que puedan probarse por completo.

- Varios tipos de clientes pueden utilizar la misma lógica de dominio.
- En el caso de los componentes de la interfaz de usuario y la base de datos son necesarias actualizaciones tecnológicas periódicas que no afectan a la lógica de la aplicación.
- Para la aplicación son necesarios varios proveedores de entrada y consumidores de salida, y la personalización de la lógica de la aplicación genera complejidad y falta de extensibilidad del código.

Problemas y consideraciones

- Diseño basado en dominios: la arquitectura hexagonal funciona muy bien con el diseño basado en dominios (DDD). Cada componente de la aplicación representa un subdominio en la DDD, y las arquitecturas hexagonales se pueden utilizar para lograr un acoplamiento débil entre los componentes de la aplicación.
- Capacidad de hacer pruebas. Desde el diseño, una arquitectura hexagonal utiliza abstracciones para las entradas y las salidas. Por lo tanto, resulta más fácil escribir las pruebas unitarias y las pruebas de manera aislada debido al acoplamiento débil inherente.
- Complejidad. La complejidad de separar la lógica empresarial del código de infraestructura, si se gestiona con cuidado, puede aportar ventajas excelentes, como la agilidad, la cobertura de las pruebas y la adaptabilidad de la tecnología. De lo contrario, la solución de los problemas puede llegar a ser compleja.
- Sobrecarga del mantenimiento. El código adaptador adicional que hace que la arquitectura sea conectable se justifica solo si el componente de la aplicación requiere varias fuentes de entrada y destinos de salida para escribir, o cuando el almacén de datos de entrada y salida tiene que cambiar con el tiempo. De lo contrario, el adaptador se convierte en una capa adicional que hay que mantener, lo que supone una sobrecarga de mantenimiento.
- Problemas de latencia. El uso de puertos y adaptadores agrega otra capa, lo que puede provocar latencia.

Implementación

Las arquitecturas hexagonales permiten que la lógica empresarial y de las aplicaciones se aíslen del código de infraestructura y del código que integra la aplicación con las interfaces de usuario, las API externas, las bases de datos y los agentes de mensajes. Puede conectar de manera fácil

los componentes de la lógica empresarial con otros componentes (como las bases de datos) de la arquitectura de la aplicación mediante puertos y adaptadores.

Los puertos son puntos de entrada independientes de la tecnología a un componente de la aplicación. Estas interfaces personalizadas determinan la interfaz que permite a los agentes externos comunicarse con el componente de la aplicación, sin importar quién o qué implemente la interfaz. Esto es similar a cómo permite un puerto USB que muchos tipos de dispositivos se comuniquen con un equipo, siempre que utilicen un adaptador USB.

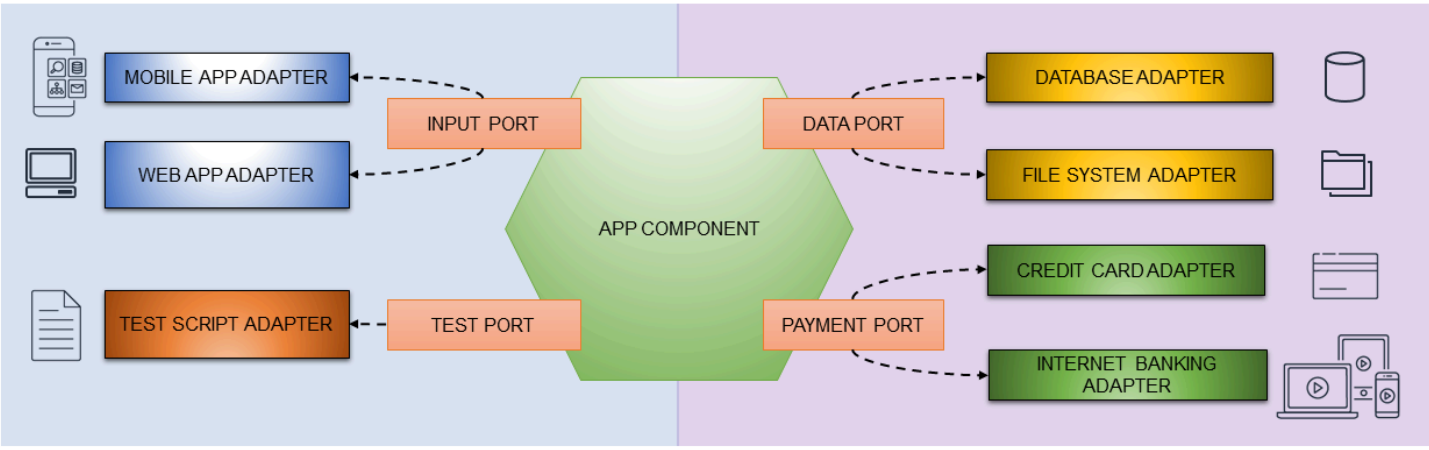
Los adaptadores interactúan con la aplicación a través de un puerto con una tecnología específica. Los adaptadores se conectan a estos puertos, reciben datos de los puertos o los proporcionan y los transforman para su procesamiento posterior. Por ejemplo, un adaptador de REST permite a los agentes comunicarse con el componente de la aplicación a través de una API de REST. Un puerto puede tener varios adaptadores sin que esto suponga ningún riesgo para el puerto ni para el componente de la aplicación. Para ampliar el ejemplo anterior, agregar un adaptador de GraphQL al mismo puerto proporciona un medio más para que los agentes interactúen con la aplicación a través de una API de GraphQL sin afectar a la API de REST, el puerto o la aplicación.

Los puertos se conectan a la aplicación y los adaptadores sirven de conexión con el mundo exterior. Puede utilizar los puertos para crear componentes de aplicaciones con acoplamiento débil e intercambiar los componentes dependientes mediante un cambio de adaptador. Esto permite que el componente de la aplicación interactúe con entradas y salidas externas sin necesidad de conocer el contexto. Los componentes son intercambiables a cualquier nivel, lo que facilita las pruebas automatizadas. Puede probar los componentes de manera independiente sin depender del código de la infraestructura, en lugar de aprovisionar un entorno completo para hacer las pruebas. La lógica de la aplicación no depende de factores externos, por lo que se simplifican las pruebas y resulta más fácil simular las dependencias.

Por ejemplo, en una arquitectura con acoplamiento débil, un componente de la aplicación debe poder leer y escribir datos sin conocer los detalles del almacén de datos. La responsabilidad del componente de la aplicación es suministrar los datos a una interfaz (puerto). Un adaptador define la lógica de escritura en un almacén de datos, que puede ser una base de datos, un sistema de archivos o un sistema de almacenamiento de objetos como Amazon S3, según las necesidades de la aplicación.

Arquitectura de alto nivel

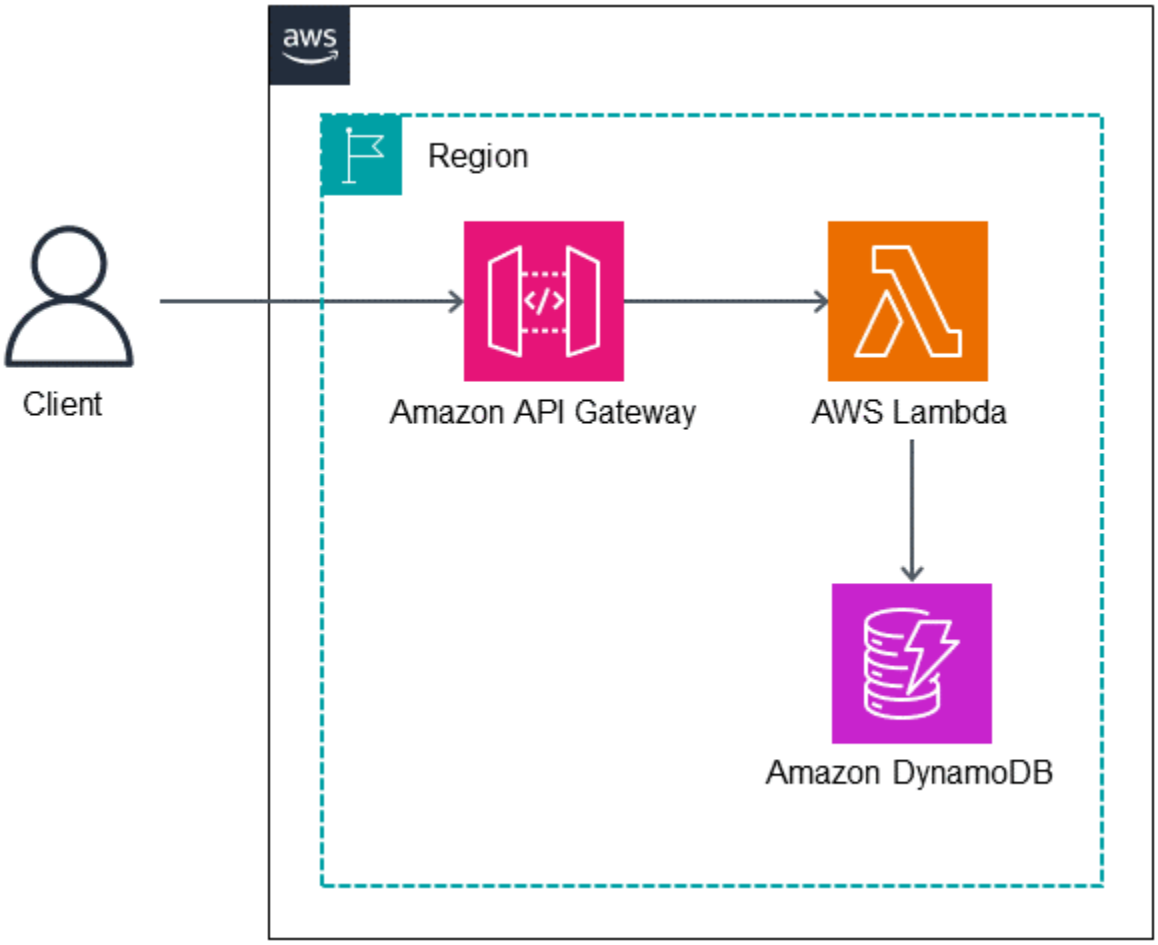
La aplicación o el componente de la aplicación contienen la lógica empresarial principal. Recibe comandos o consultas de los puertos y envía las solicitudes a través de los puertos a agentes externos, que se implementan mediante adaptadores, como se ilustra en el diagrama siguiente.



Implementación mediante Servicios de AWS

Las funciones de AWS Lambda suelen contener lógica empresarial y código de integración de bases de datos, que están acoplados de manera ajustada para cumplir un objetivo. Puede utilizar el patrón de arquitectura hexagonal para separar la lógica empresarial del código de infraestructura. Esta separación permite hacer pruebas unitarias de la lógica empresarial sin depender del código de la base de datos y mejora la agilidad del proceso de desarrollo.

En la arquitectura siguiente, una función de Lambda implementa el patrón de arquitectura hexagonal. La API de REST de Amazon API Gateway inicia la función de Lambda. La función implementa la lógica empresarial y escribe los datos en las tablas de DynamoDB.



Código de muestra

El código de ejemplo de esta sección muestra cómo implementar el modelo de dominio mediante Lambda, separarlo del código de infraestructura (como el código para acceder a DynamoDB) e implementar las pruebas unitarias para la función.

Modelo de dominio

La clase de modelo de dominio no conoce las dependencias ni los componentes externos, solo implementa la lógica empresarial. En el ejemplo siguiente, la clase `Recipient` es una clase de modelo de dominio que verifica si hay superposiciones en la fecha de reserva.

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

Puerto de entrada

La clase `RecipientInputPort` se conecta con la clase receptora y ejecuta la lógica del dominio.

```
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
```

Clase del adaptador de DynamoDB

La clase `DDBRecipientAdapter` implementa el acceso a las tablas de DynamoDB.

```
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": []
            }
          # ...
```

La función de Lambda `get_recipient_input_port` es una fábrica de instancias de la clase `RecipientInputPort`. Construye instancias de las clases de puertos de salida con las instancias del adaptador relacionadas.

```
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
        }),
    }
```

Pruebas unitarias

Puede probar la lógica empresarial de las clases de modelos de dominio mediante la inyección de clases simuladas. En el ejemplo siguiente se proporciona la prueba unitaria de la clase `Recipent` del modelo de dominio.

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

Repositorio GitHub

Para obtener una implementación completa de la arquitectura de ejemplo para este patrón, consulte el repositorio de GitHub en https://github.com/aws-samples/aws-lambda-domain-model-sample.

Contenido relacionado

- Hexagonal architecture, artículo de Alistair Cockburn
- Developing evolutionary architectures with AWS Lambda (AWS entrada del blog en japonés)

Videos

En el video siguiente (en japonés) se analiza el uso de la arquitectura hexagonal en la implementación de un modelo de dominio mediante una función de Lambda.

Patrón de publicación/suscripción

Intención

El patrón de publicación/suscripción (también conocido como patrón pub/sub) es un patrón de mensajería que desvincula al remitente de un mensaje (publicador) de los receptores interesados (suscriptores). Este patrón implementa las comunicaciones asíncronas mediante la publicación de mensajes o eventos a través de un intermediario conocido como agente de mensajes o enrutador (infraestructura de mensajes). El patrón de publicación/suscripción aumenta la escalabilidad y la capacidad de respuesta de los remitentes al delegar la responsabilidad de la entrega del mensaje en la infraestructura de mensajes, de modo que el remitente puede centrarse en el procesamiento principal de los mensajes.

Motivación

En las arquitecturas distribuidas, los componentes del sistema a menudo necesitan proporcionar información a otros componentes a medida que los eventos tienen lugar dentro del sistema. El patrón de publicación/suscripción separa las preocupaciones para que las aplicaciones puedan centrarse en sus capacidades principales, mientras que la infraestructura de mensajes se encarga de las responsabilidades de comunicación, como el enrutamiento de los mensajes y la entrega confiable. El patrón de publicación/suscripción permite que la mensajería asíncrona desvincule al publicador de los suscriptores. Los publicadores también pueden enviar mensajes sin que los suscriptores lo sepan.

Aplicabilidad

Utilice el patrón de publicación/suscripción cuando:

- El procesamiento paralelo es necesario si un solo mensaje tiene flujos de trabajo diferentes.
- No es necesario transmitir mensajes a varios suscriptores ni responder en tiempo real por parte de los receptores.
- El sistema o la aplicación pueden tolerar la posible coherencia de los datos o el estado.
- La aplicación o el componente tiene que comunicarse con otras aplicaciones o servicios que pueden utilizar diferentes lenguajes, protocolos o plataformas.

Problemas y consideraciones

- Disponibilidad de los suscriptores: el publicador no sabe si los suscriptores están escuchando o no. Los mensajes publicados son de naturaleza transitoria y pueden provocar que se eliminen si los suscriptores no están disponibles.
- Garantía de entrega de mensajes: por lo general, el patrón de publicación/suscripción no garantiza la entrega de mensajes a todos los tipos de suscriptores, aunque algunos servicios, como Amazon Simple Notification Service (Amazon SNS), pueden proporcionar la entrega única a algunos subconjuntos de suscriptores.
- Tiempo de vida (TTL): los mensajes tienen una vida útil y caducan si no se procesan dentro de ese periodo de tiempo. Considere la posibilidad de agregar los mensajes publicados a una cola para que puedan conservarse y garantizar que se procesen más allá del periodo de TTL.
- Relevancia de los mensajes: los productores pueden establecer un intervalo de tiempo de relevancia como parte de los datos del mensaje, y el mensaje puede descartarse después de esa fecha. Considere la posibilidad de diseñar a los consumidores para que examinen esta información antes de decidir cómo procesar el mensaje.
- Coherencia final: hay un retraso entre el momento en que se publica el mensaje y el momento en que lo consume el suscriptor. Esto podría provocar que los almacenes de datos de los suscriptores acaben siendo coherentes cuando se requiera una coherencia sólida. La coherencia final también puede ser un problema cuando los productores y los consumidores requieren una interacción casi en tiempo real.
- Comunicación unidireccional: el patrón de publicación/suscripción se considera unidireccional. Las aplicaciones que requieren mensajería bidireccional con un canal de suscripción de retorno deberían considerar la posibilidad de utilizar un patrón de solicitud y respuesta si se requiere una respuesta sincrónica.
- Orden de los mensajes: no se garantiza el orden de los mensajes. Si los consumidores necesitan mensajes ordenados, le recomendamos que utilice los temas FIFO de Amazon SNS para garantizar el orden.
- Duplicación de mensajes: según la infraestructura de mensajería, se pueden entregar mensajes duplicados a los consumidores. Los consumidores deben estar diseñados para ser idempotentes a la hora de gestionar el procesamiento de mensajes duplicados. Como alternativa, utilice los temas FIFO de Amazon SNS para garantizar una entrega única.
- Filtrado de mensajes: los consumidores suelen estar interesados únicamente en un subconjunto de los mensajes publicados por un productor. Proporcione mecanismos que permitan a los

suscriptores filtrar o restringir los mensajes que reciben proporcionando filtros de contenido o temas.

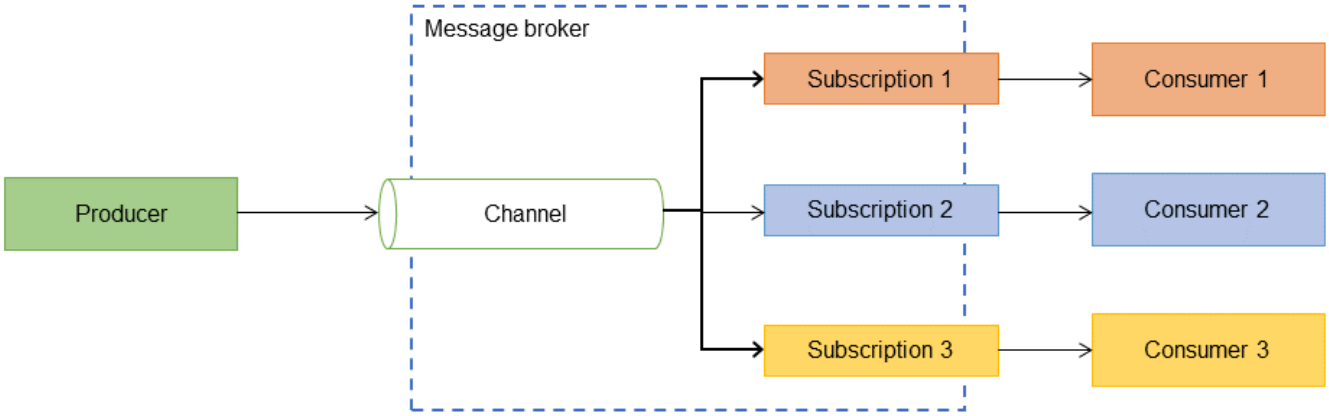
- Reproducción de mensajes: las capacidades de reproducción de mensajes pueden depender de la infraestructura de mensajería. También puede proporcionar implementaciones personalizadas según el caso de uso.
- Colas de mensajes fallidos: en un sistema postal, una oficina de mensajes fallidos es una instalación para procesar el correo que no se puede entregar. En la mensajería publicación/suscripción, una cola de mensajes fallidos (DLQ) es una cola de mensajes que no se pueden entregar a un punto de conexión suscrito.

Implementación

Arquitectura de alto nivel

En un patrón de publicación/suscripción, el subsistema de mensajería asíncrona, conocido como enrutador o agente de mensajes, realiza un seguimiento de las suscripciones. Cuando un productor publica un evento, la infraestructura de mensajería envía un mensaje a cada consumidor. Una vez que se envía un mensaje a los suscriptores, se elimina de la infraestructura de mensajes para que no se pueda reproducir y los nuevos suscriptores no ven el evento. Los enrutadores o agentes de mensajes separan al productor de eventos de los consumidores de mensajes de la siguiente manera:

- Proporcionan un canal de entrada para que el productor publique eventos empaquetados en mensajes, utilizando un formato de mensaje definido.
- Crean un canal de salida individual por suscripción. Una suscripción es la conexión del consumidor, donde escucha los mensajes de eventos que están asociados a un canal de entrada específico.
- Copian los mensajes del canal de entrada al canal de salida para todos los consumidores cuando se publica el evento.



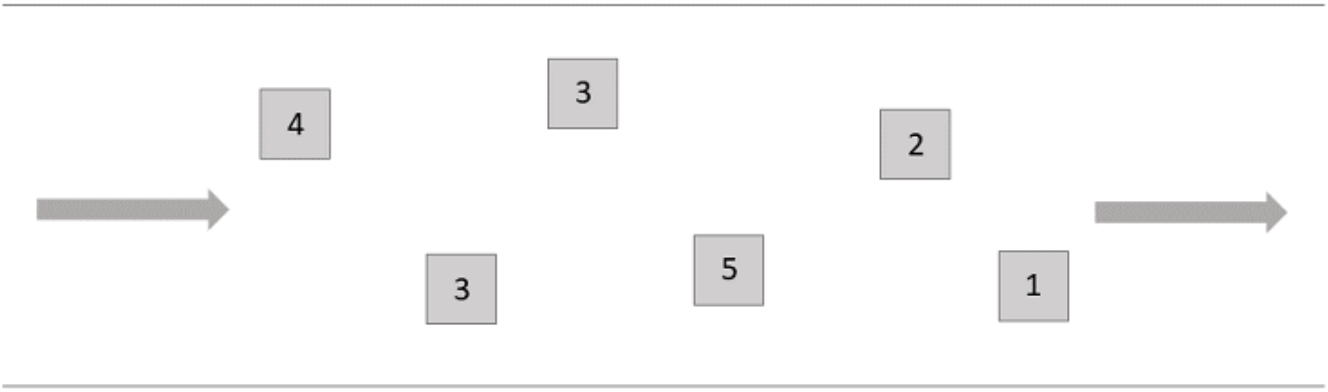
Implementación mediante los servicios de AWS

Amazon SNS

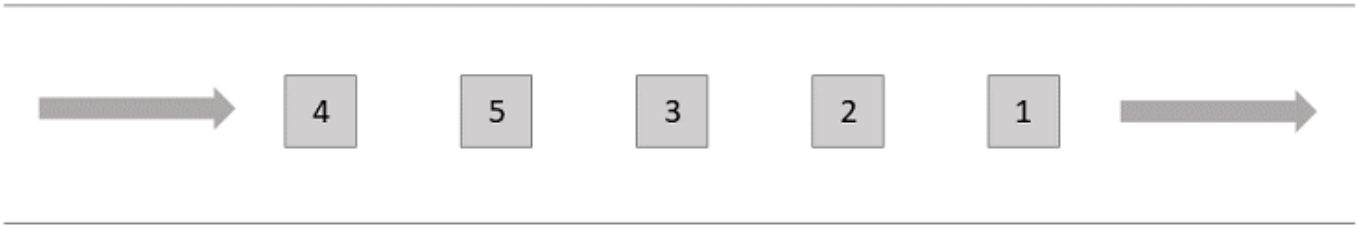
Amazon SNS es un servicio publicador/suscriptor totalmente administrado que proporciona mensajería de aplicación a aplicación (A2A) para desvincular aplicaciones distribuidas. También proporciona mensajería de aplicación a persona (A2P) para enviar SMS, correo electrónico y otras notificaciones push.

Amazon SNS ofrece dos tipos de temas: estándar y primero en entrar, primero en salir (FIFO).

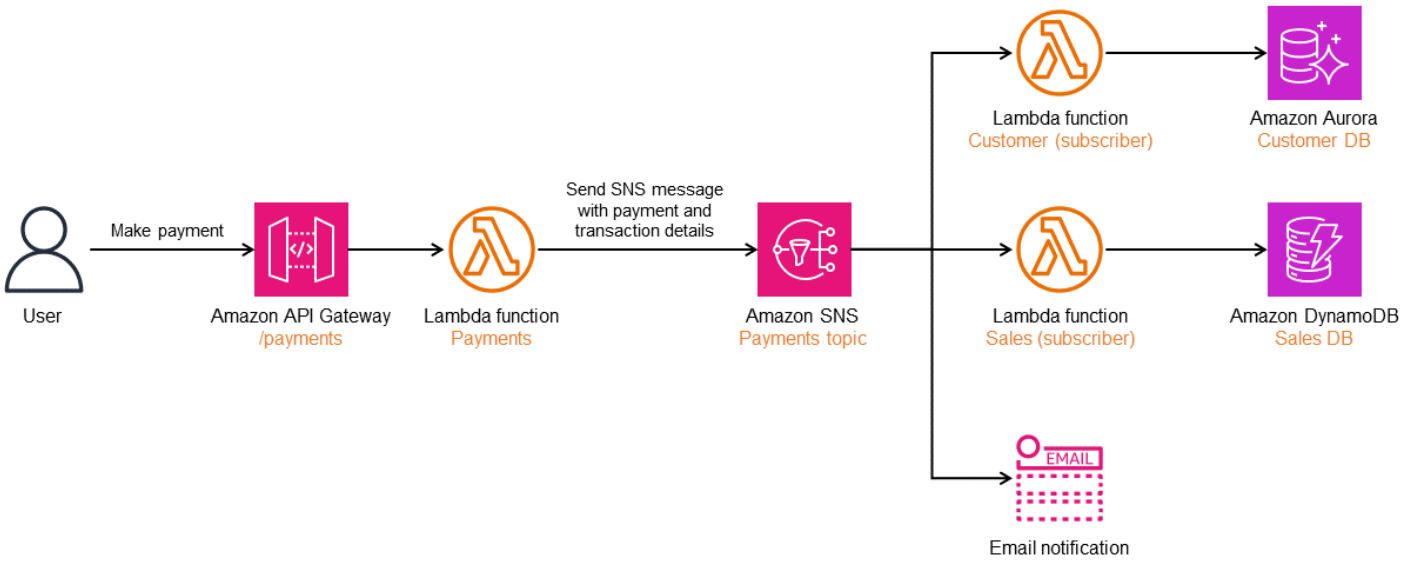
- Los temas estándar admiten un número ilimitado de mensajes por segundo y ofrecen la mejor forma de ordenar y deduplicar.



- Los temas FIFO proporcionan un orden y una deduplicación estrictos, y admiten hasta 300 mensajes por segundo o 10 MB por segundo por tema de FIFO (lo que ocurra primero).



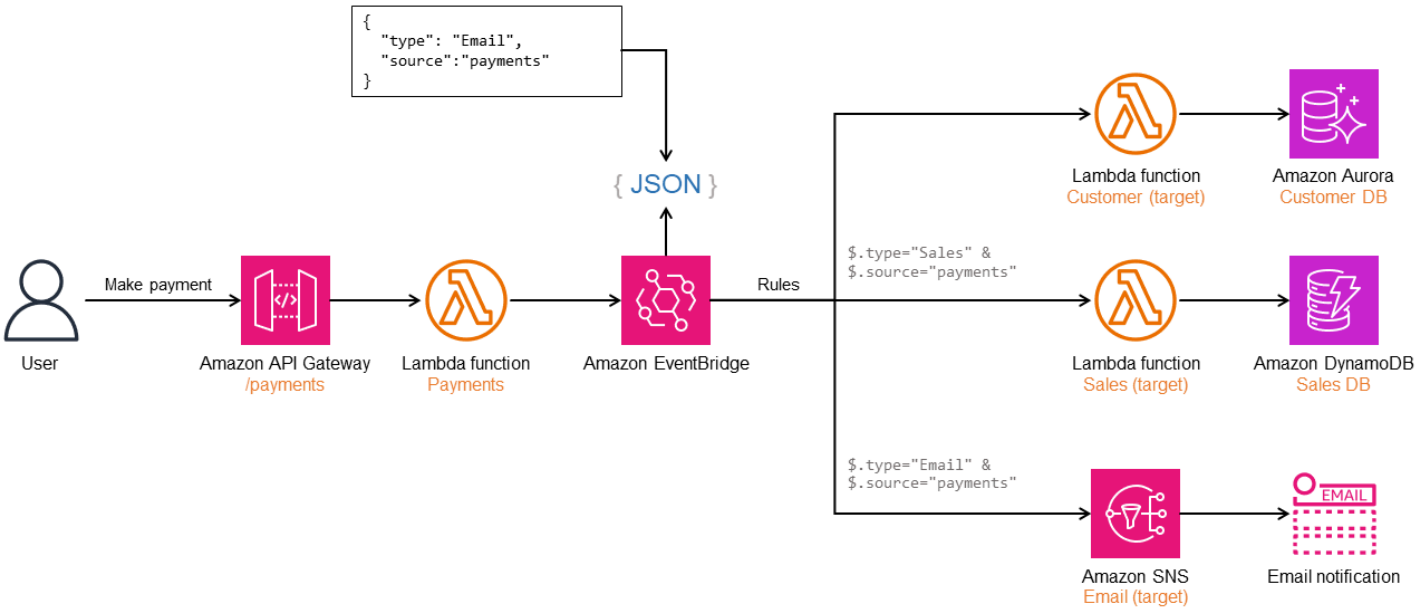
La siguiente ilustración muestra cómo puede utilizar Amazon SNS para implementar el patrón de publicación/suscripción. Cuando un usuario realiza un pago, la función de Lambda `Payments` envía un mensaje de SNS al tema de SNS `Payments`. Este tema de SNS tiene tres suscriptores. Cada suscriptor recibe una copia del mensaje y la procesa.



Amazon EventBridge

Puede usar Amazon EventBridge cuando necesite un enrutamiento más complejo de mensajes de varios productores a través de diferentes protocolos a consumidores suscritos o suscripciones directas y distribuidas. EventBridge también admite el enrutamiento, el filtrado, la secuenciación y la división o agregación basados en contenido. En la siguiente ilustración, EventBridge se utiliza para crear una versión del patrón de publicación/suscripción en el que los suscriptores se definen mediante reglas de eventos. Cuando un usuario realiza un pago, la función de Lambda `Payments` envía un mensaje a EventBridge mediante el bus de eventos predeterminado basado en un esquema

personalizado que tiene tres reglas que apuntan a destinos diferentes. Cada microservicio procesa los mensajes y realiza las acciones necesarias.



Taller

- Creación de arquitecturas basadas en eventos en AWS
- Envío de notificaciones de eventos de distribución ramificada con Amazon Simple Queue Service (Amazon SQS) y Amazon Simple Notification Service (Amazon SNS)

Referencias de blogs

- Elección entre servicios de mensajería para aplicaciones sin servidor
- Diseño de aplicaciones duraderas sin servidor con DLQ para Amazon SNS, Amazon SQS y AWS Lambda
- Simplificación de los mensajes de publicación/suscripción con el filtrado de mensajes de Amazon SNS

Contenido relacionado

- Características de mensajería de publicación/suscripción

Patrón de reintento con retroceso

Intención

El patrón de reintento con retroceso mejora la estabilidad de la aplicación al reintentar de manera transparente las operaciones con fallas debido a errores transitorios.

Motivación

En las arquitecturas distribuidas, los errores transitorios podrían deberse a la limitación del servicio, la pérdida temporal de la conectividad de la red o la no disponibilidad temporal del servicio. Reintentar de manera automática las operaciones con fallas debido a estos errores transitorios mejora la experiencia del usuario y la resiliencia de las aplicaciones. Sin embargo, los reintentos frecuentes pueden sobrecargar el ancho de banda de la red y provocar una contención. El retroceso exponencial es una técnica en la que las operaciones se reintentan mediante el aumento de los tiempos de espera para un número específico de reintentos.

Aplicabilidad

Utilice el patrón de reintento con retroceso cuando suceda lo siguiente:

- Los servicios suelen limitar las solicitudes para evitar la sobrecarga, lo que se traduce en una excepción al proceso de llamada 429: demasiadas solicitudes.
- La red es un participante invisible en las arquitecturas distribuidas, y los problemas temporales de la red provocan fallas.
- El servicio al que se llama no está disponible de manera temporal, lo que provoca errores. Los reintentos frecuentes pueden provocar una degradación del servicio, a menos que se indique un tiempo de espera mediante este patrón.

Problemas y consideraciones

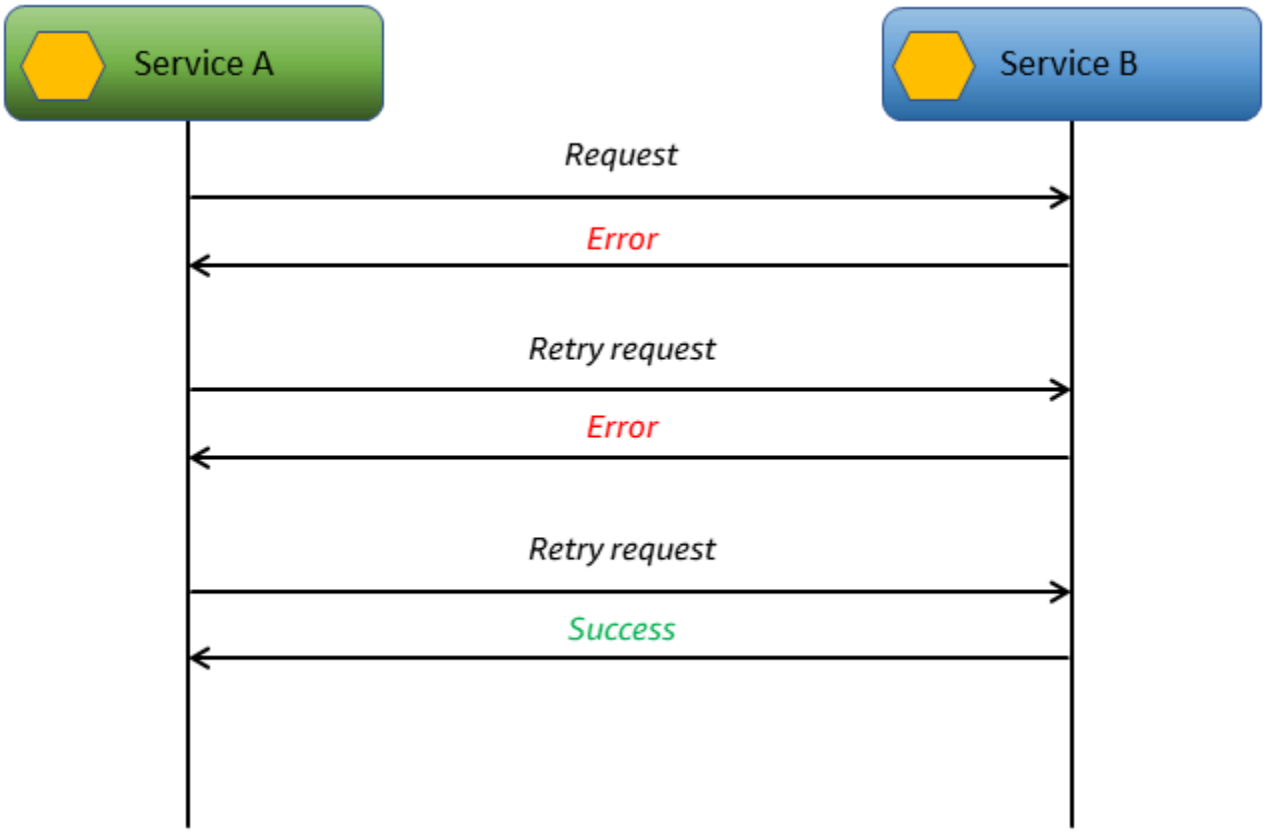
- Idempotencia: si varias llamadas al método tienen el mismo efecto que una sola llamada en el estado del sistema, la operación se considera idempotente. Las operaciones deben ser idempotentes cuando se utiliza el patrón de reintento con retroceso. De lo contrario, las actualizaciones parciales podrían dañar el estado del sistema.

- Ancho de banda de la red: se puede producir una degradación del servicio si demasiados reintentos ocupan el ancho de banda de la red, lo que reduce los tiempos de respuesta.
- Escenarios de respuesta rápida a los errores: en el caso de errores no transitorios, si se puede determinar la causa del error, es más eficiente responder rápido a los errores mediante el patrón de disyuntores.
- Tasa de retroceso: la introducción de un retroceso exponencial puede repercutir en el tiempo de espera del servicio, lo que se traduce en tiempos de espera más largos para el usuario final.

Implementación

Arquitectura de alto nivel

En el diagrama siguiente se ilustra cómo el servicio A puede reintentar las llamadas al servicio B hasta obtener una respuesta correcta. Si el servicio B no devuelve una respuesta satisfactoria después de varios intentos, el servicio A puede dejar de reintentarlo y devolver un error al autor de la llamada.

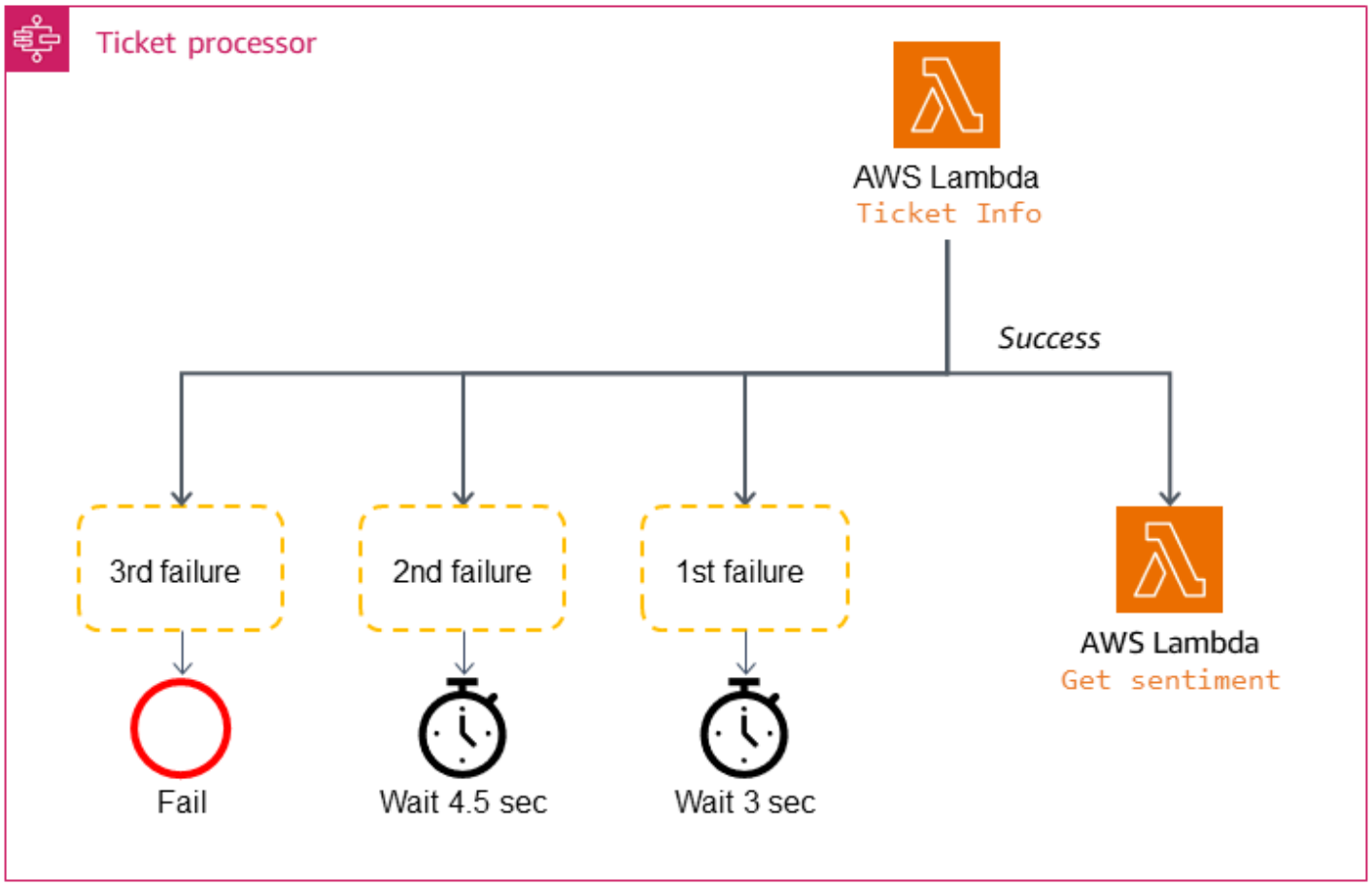


Implementación mediante los servicios de AWS

En el diagrama siguiente se muestra un flujo de trabajo del procesamiento de tickets en una plataforma de atención al cliente. Los tickets de los clientes insatisfechos se gestionan con mayor rapidez mediante el escalado automático de su prioridad. La función de Lambda `Ticket info` extrae los detalles del ticket y llama a la función de Lambda `Get sentiment`. La función de Lambda `Get sentiment` verifica las opiniones de los clientes al pasar la descripción a Amazon Comprehend (no se muestra).

Si se produce un error en la llamada a la función de Lambda `Get sentiment`, el flujo de trabajo vuelve a intentar la operación tres veces. AWS Step Functions permite el retroceso exponencial al permitirle configurar el valor del retroceso.

En este ejemplo, se configuran un máximo de tres reintentos con un multiplicador de aumento de 1,5 segundos. Si el primer reintento se produce después de 3 segundos, el segundo se produce después de 3 x 1,5 segundos = 4,5 segundos y el tercer reintento se produce después de 4,5 x 1,5 segundos = 6,75 segundos. Si el tercer reintento es correcto, se produce un error en el flujo de trabajo. Para la lógica de retroceso no es necesario ningún código personalizado. AWS Step Functions lo proporciona como una configuración.



Código de muestra

En el código siguiente se muestra la implementación del patrón de reintento con retroceso.

```
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);
  }
```

Repositorio GitHub

Para obtener una implementación completa de la arquitectura de ejemplo para este patrón, consulte el repositorio de GitHub en https://github.com/aws-samples/retry-with-backoff.

Contenido relacionado

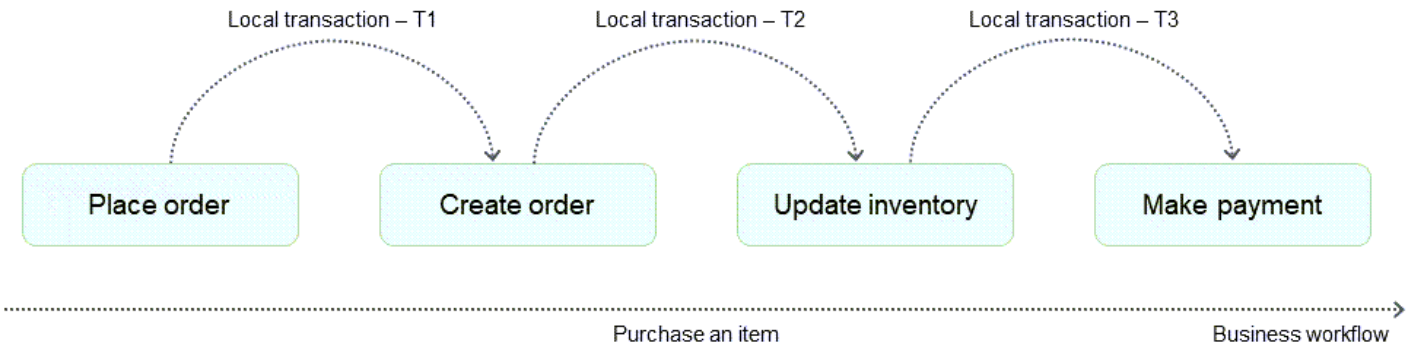
- Timeouts, retries, and backoff with jitter (Amazon Builders' Library)

Patrones saga

Una saga consiste en una secuencia de transacciones locales. Cada transacción local de una saga actualiza la base de datos y desencadena la siguiente transacción local. Si se produce un error en una transacción, la saga ejecuta transacciones de compensación para revertir los cambios en la base de datos realizados por las transacciones anteriores.

Esta secuencia de transacciones locales ayuda a lograr un flujo de trabajo empresarial al utilizar los principios de continuidad y compensación. El principio de continuación decide la recuperación anticipada del flujo de trabajo, mientras que el principio de compensación decide la recuperación hacia atrás. Si se produce un error en la actualización en algún paso de la transacción, la saga publica un evento para continuar (para volver a intentar la transacción) o como compensación (para volver al estado anterior de los datos). Esto garantiza que la integridad de los datos se mantenga y sea coherente en todos los almacenes de datos.

Por ejemplo, cuando un usuario compra un libro en una tienda online, el proceso consiste en una secuencia de transacciones, como la creación de un pedido, la actualización del inventario, el pago y el envío, que representa un flujo de trabajo empresarial. Para completar este flujo de trabajo, la arquitectura distribuida emite una secuencia de transacciones locales para crear un pedido en la base de datos de pedidos, actualizar la base de datos de inventario y actualizar la base de datos de pagos. Cuando el proceso se realiza correctamente, estas transacciones se invocan secuencialmente para completar el flujo de trabajo empresarial, como se muestra en el siguiente diagrama. Sin embargo, si se produce un error en alguna de estas transacciones locales, el sistema debería poder decidir cuál es el siguiente paso adecuado, es decir, una recuperación hacia adelante o hacia atrás.



Los dos escenarios siguientes ayudan a determinar si el siguiente paso es la recuperación hacia delante o hacia atrás:

- Error a nivel de plataforma, en el que algo falla en la infraestructura subyacente y provoca un error en la transacción. En este caso, el patrón saga puede provocar una recuperación hacia delante al volver a intentar la transacción local y continuar con el proceso empresarial.
- Error a nivel de aplicación, en el que el servicio de pago falla debido a un pago no válido. En este caso, el patrón saga puede realizar una recuperación hacia atrás mediante la emisión de una transacción de compensación para actualizar el inventario y las bases de datos de pedidos y restablecer su estado anterior.

El patrón saga gestiona el flujo de trabajo empresarial y garantiza que se alcance un estado final deseable mediante la recuperación hacia delante. En caso de errores, revierte las transacciones locales mediante la recuperación hacia atrás para evitar problemas de coherencia de datos.

El patrón saga tiene dos variantes: coreografía y orquestación.

Coreografía de la saga

El patrón de coreografía de la saga depende de los eventos publicados por los microservicios. Los participantes de la saga (microservicios) se suscriben a los eventos y actúan en función de los factores desencadenantes de los eventos. Por ejemplo, el servicio de pedidos del siguiente diagrama emite un evento `OrderPlaced`. El servicio de inventario se suscribe a ese evento y actualiza el inventario cuando se emite el evento `OrderPlaced`. Del mismo modo, los servicios participantes actúan en función del contexto del evento emitido.

El patrón de coreografía de la saga es adecuado cuando solo hay unos pocos participantes en la saga y se necesita una implementación sencilla sin ningún punto de error. Cuando se agregan más participantes, se hace más difícil realizar un seguimiento de las dependencias entre los participantes mediante este patrón.

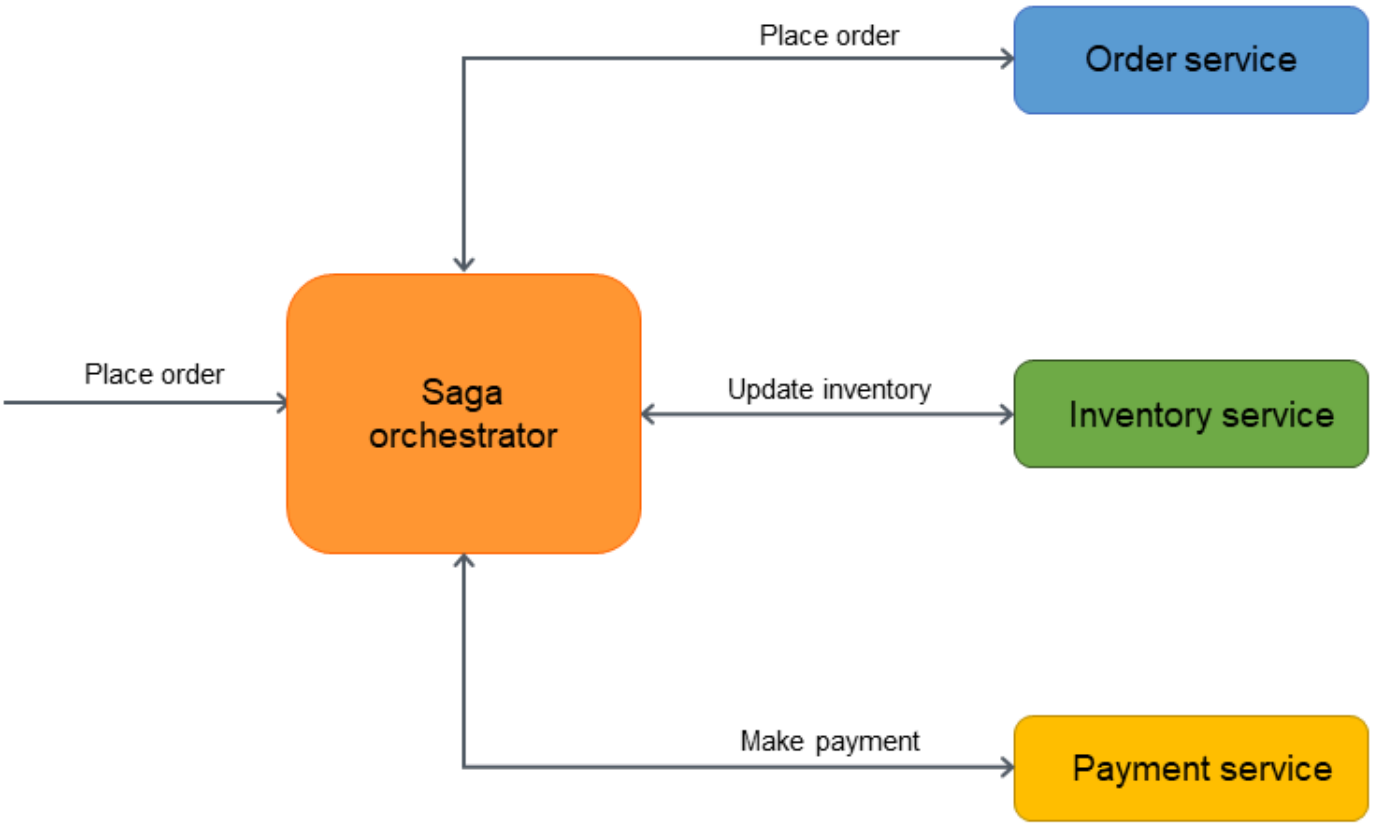
Para una revisión detallada, consulte la sección Coreografía de la saga de esta guía.

Orquestación de la saga

El patrón de orquestación de la saga tiene un coordinador central llamado orquestador. El orquestador de la saga administra y coordina todo el ciclo de vida de la transacción. Conoce la

serie de pasos que se deben realizar para completar la transacción. Para ejecutar un paso, envía un mensaje al microservicio participante para que realice la operación. El microservicio participante completa la operación y envía un mensaje al orquestador. En función del mensaje que recibe, el orquestador decide qué microservicio ejecutar a continuación en la transacción.

El patrón de orquestación de la saga es adecuado cuando hay muchos participantes, y es necesario un acoplamiento flexible entre los participantes de la saga. El orquestador resume la complejidad de la lógica al realizar el acoplamiento flexible de los participantes. Sin embargo, el orquestador puede convertirse en un único punto de error porque controla todo el flujo de trabajo.



Para una revisión detallada, consulte la sección Orquestación de la saga de esta guía.

Patrón de coreografía de la saga

Intención

El patrón de coreografía de la saga ayuda a preservar la integridad de los datos en las transacciones distribuidas que abarcan varios servicios mediante el uso de suscripciones a eventos. En una transacción distribuida, se puede llamar a varios servicios antes de que se complete la transacción.

Cuando los servicios almacenan datos en diferentes almacenes de datos, puede resultar difícil mantener la coherencia de datos entre estos almacenes de datos.

Motivación

Una transacción es una sola unidad de trabajo que puede implicar varios pasos, donde todos los pasos se ejecutan por completo o no se ejecuta ningún paso, lo que da como resultado un almacén de datos que conserva su estado coherente. Los términos atomicidad, coherencia, aislamiento y durabilidad (ACID, por sus siglas en inglés) definen las propiedades de una transacción. Las bases de datos relacionales proporcionan transacciones ACID para mantener la coherencia de datos.

Para mantener la coherencia en una transacción, las bases de datos relacionales utilizan el método de confirmación en dos fases (2PC). Consiste en una fase de preparación y una fase de confirmación.

- En la fase de preparación, el proceso de coordinación solicita a los procesos participantes en la transacción (participantes) que se comprometan a confirmar o anular la transacción.
- En la fase de confirmación, el proceso de coordinación solicita a los participantes que confirmen la transacción. Si los participantes no pueden ponerse de acuerdo para realizar la confirmación en la fase de preparación, la transacción se anula.

En los sistemas distribuidos que siguen un patrón de diseño de base de datos por servicio, la confirmación en dos fases no es una opción. Esto se debe a que cada transacción se distribuye en varias bases de datos y no existe un único controlador que pueda coordinar un proceso similar a la confirmación en dos fases de los almacenes de datos relacionales. En este caso, una solución es utilizar el patrón de coreografía de la saga.

Aplicabilidad

Use el patrón de coreografía de la saga cuando:

- El sistema requiere integridad y coherencia de los datos en las transacciones distribuidas que abarcan varios almacenes de datos.
- El almacén de datos (por ejemplo, una base de datos NoSQL) no proporciona la confirmación en dos fases para proporcionar transacciones ACID. Es necesario actualizar varias tablas dentro de una sola transacción e implementar la confirmación en dos fases dentro de los límites de la aplicación sería una tarea compleja.

- Un proceso de control central que administra las transacciones de los participantes podría convertirse en un único punto de error.
- Los participantes de la saga son servicios independientes y deben tener un acoplamiento flexible.
- Existe comunicación entre contextos delimitados en un dominio empresarial.

Problemas y consideraciones

- Complejidad: a medida que aumenta el número de microservicios, la coreografía de la saga puede resultar difícil de administrar debido a la cantidad de interacciones entre los microservicios. Además, las transacciones y los reintentos de compensación agregan complejidad al código de la aplicación, lo que puede provocar una sobrecarga de mantenimiento. La coreografía es adecuada cuando solo hay unos pocos participantes en la saga y se necesita una implementación sencilla sin ningún punto de error. Cuando se agregan más participantes, se hace más difícil realizar un seguimiento de las dependencias entre los participantes mediante este patrón.
- Implementación resiliente: en la coreografía de la saga, es más difícil implementar los tiempos de espera, los reintentos y otros patrones de resiliencia a nivel mundial que en la orquestación de la saga. La coreografía debe implementarse en componentes individuales y no a nivel de un orquestador.
- Dependencias cíclicas: los participantes consumen mensajes que se publican unos a otros. Esto podría dar lugar a dependencias cíclicas, lo que generaría complejidades en el código y sobrecargas de mantenimiento, además de posibles bloqueos.
- Problema de escritura doble: el microservicio tiene que actualizar atómicamente la base de datos y publicar un evento. El error de cualquiera de las dos operaciones puede provocar un estado incoherente. Una forma de solucionar este problema es utilizar el patrón de bandeja de salida transaccional.
- Conservación de los eventos: los participantes de la saga actúan en función de los eventos publicados. Es importante guardar los eventos en el orden en que se producen para poder auditarlos, depurarlos y reproducirlos. Puede utilizar el patrón de aprovisionamiento de eventos para conservar los eventos en un almacén de eventos en caso de que sea necesario reproducir el estado del sistema para restablecer la coherencia de datos. Los almacenes de eventos también se pueden utilizar con fines de auditoría y solución de problemas, ya que reflejan todos los cambios en el sistema.
- Coherencia final: el procesamiento secuencial de las transacciones locales da como resultado una coherencia final, lo que puede ser un desafío en los sistemas que requieren una gran coherencia. Para abordar este problema, establezca las expectativas de sus equipos empresariales con

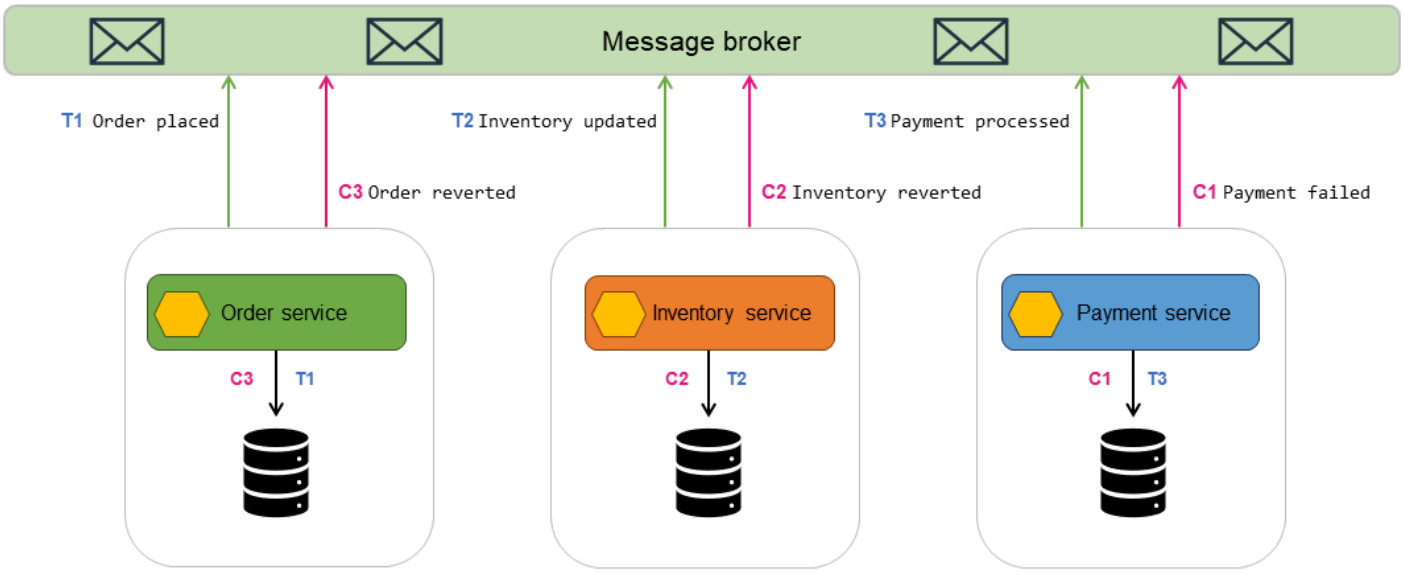
respecto al modelo de coherencia o reevalúe el caso de uso y cambie a una base de datos que ofrezca una coherencia sólida.

- Idempotencia: los participantes de la saga tienen que ser idempotentes para permitir la ejecución repetida en caso de que se produzcan errores transitorios provocados por bloqueos inesperados o errores del orquestador.
- Aislamiento de transacciones: el patrón saga carece de aislamiento de transacciones, que es una de las cuatro propiedades de las transacciones ACID. El grado de aislamiento de una transacción determina el grado en el que otras transacciones simultáneas pueden afectar a los datos en los que opera. La orquestación simultánea de las transacciones puede provocar datos obsoletos. Recomendamos utilizar el bloqueo semántico para gestionar estos escenarios.
- Observabilidad: la observabilidad se refiere al registro y el seguimiento detallados para solucionar problemas en el proceso de implementación y orquestación. Esto se vuelve importante cuando aumenta el número de participantes de la saga, lo que dificulta la depuración. La supervisión y la elaboración de informes integrales son más difíciles de lograr en la coreografía de la saga que en la orquestación de la saga.
- Problemas de latencia: las transacciones compensatorias pueden agregar latencia al tiempo de respuesta total cuando la saga consta de varios pasos. Si las transacciones realizan llamadas sincrónicas, la latencia puede aumentar aún más.

Implementación

Arquitectura de alto nivel

En el siguiente diagrama de arquitectura, la coreografía de la saga tiene tres participantes: el servicio de pedidos, el servicio de inventario y el servicio de pago. Se requieren tres pasos para completar la transacción: T1, T2 y T3. Tres transacciones compensatorias restauran los datos al estado inicial: C1, C2 y C3.



- El servicio de pedidos ejecuta una transacción local, la T1, que actualiza de forma atómica la base de datos y publica un mensaje `Order placed` en el agente de mensajes.
- El servicio de inventario se suscribe a los mensajes del servicio de pedidos y recibe el mensaje de que se ha creado un pedido.
- El servicio de inventario ejecuta una transacción local, la T2, que actualiza de forma atómica la base de datos y publica un mensaje `Inventory updated` en el agente de mensajes.
- El servicio de pago se suscribe a los mensajes del servicio de inventario y recibe el mensaje de que el inventario se ha actualizado.
- El servicio de pago ejecuta una transacción local, la T3, que actualiza de forma atómica la base de datos con los detalles de los pagos y publica un mensaje `Payment processed` en el agente de mensajes.
- Si el pago no se realiza correctamente, el servicio de pago ejecuta una transacción compensatoria, la C1, que revierte el pago de forma atómica en la base de datos y publica un mensaje `Payment failed` en el agente de mensajes.
- Las transacciones compensatorias C2 y C3 se ejecutan para restablecer la coherencia de datos.

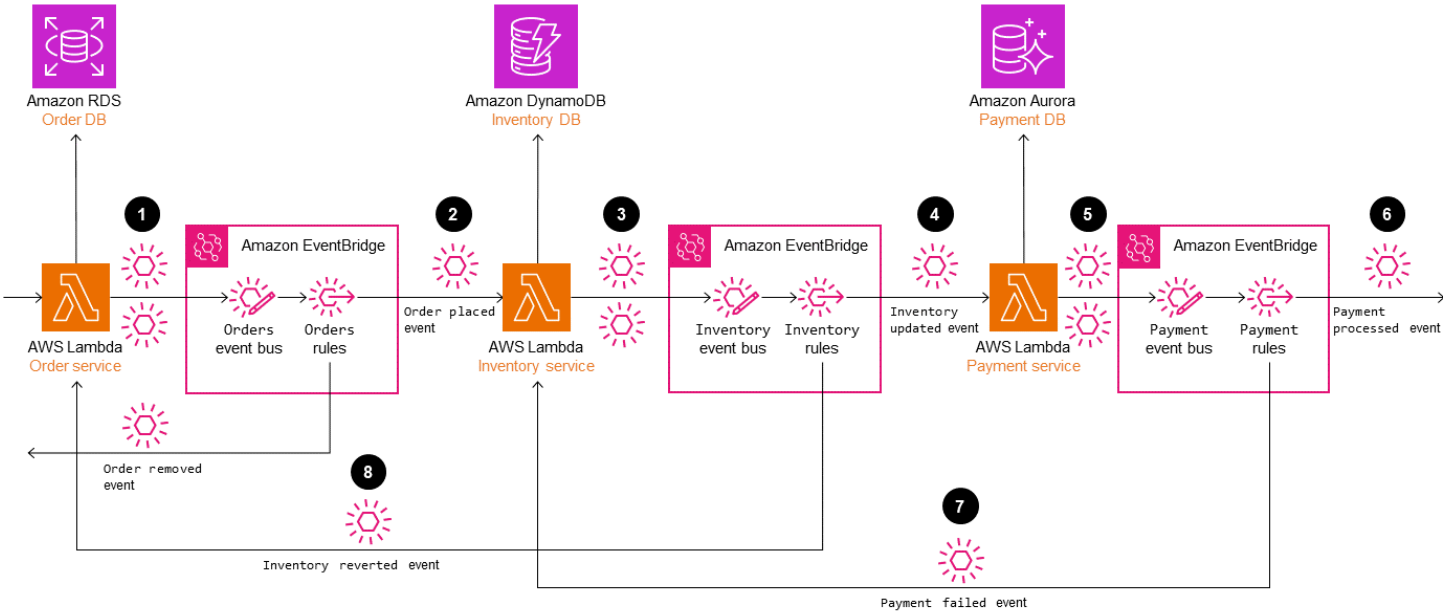
Implementación mediante los servicios de AWS

Puede implementar el patrón de coreografía de la saga mediante Amazon EventBridge. EventBridge utiliza eventos para conectar los componentes de la aplicación. Procesa los eventos a través de buses o canalizaciones de eventos. Un bus de eventos es un enrutador que recibe eventos y los

envía a cero o más destinos u objetivos. Las reglas asociadas al bus de eventos evalúan los eventos a medida que llegan y los envían a los objetivos para su procesamiento.

En la siguiente arquitectura:

- Los microservicios (servicio de pedidos, servicio de inventario y servicio de pago) se implementan como funciones de Lambda.
- Hay tres buses personalizados de EventBridge: bus de eventos `Orders`, bus de eventos `Inventory` y bus de eventos `Payment`.
- Las reglas de `Orders`, reglas de `Inventory` y reglas de `Payment` hacen coincidir los eventos que se envían al bus de eventos correspondiente e invocan las funciones de Lambda.



En un escenario que se ejecuta correctamente, cuando se realiza un pedido:

1. El servicio de pedidos procesa la solicitud y envía el evento al bus de eventos `Orders`.
2. Las reglas de `Orders` coinciden con los eventos e inician el servicio de inventario.
3. El servicio de inventario actualiza el inventario y envía el evento al bus de eventos `Inventory`.
4. Las reglas de `Inventory` coinciden con los eventos e inician el servicio de pago.
5. El servicio de pago procesa la solicitud y envía el evento al bus de eventos `Payment`.

6. Las reglas de `Payment` coinciden con los eventos y envían la notificación del evento `Payment processed` al oyente.

   Como alternativa, cuando hay un problema en el procesamiento de pedidos, las reglas de EventBridge inician las transacciones compensatorias para revertir las actualizaciones de datos y mantener la coherencia e integridad de los datos.
7. Si el pago no se realiza correctamente, las reglas de `Payment` procesan el evento e inician el servicio de inventario. El servicio de inventario ejecuta transacciones compensatorias para revertir el inventario.
8. Cuando el inventario se ha revertido, el servicio de inventario envía el evento `Inventory reverted` al bus de eventos `Inventory`. Este evento se procesa mediante reglas de `Inventory`.. Inicia el servicio de pedidos, que ejecuta la transacción de compensación para eliminar el pedido.

Contenido relacionado

- Patrón de orquestación de la saga
- Patrón de bandeja de salida transaccional
- Patrón de reintento con retroceso

Patrón de orquestación de la saga

Intención

El patrón de orquestación de la saga utiliza un coordinador central (orquestador) para ayudar a preservar la integridad de los datos en las transacciones distribuidas que abarcan varios servicios. En una transacción distribuida, se puede llamar a varios servicios antes de que se complete la transacción. Cuando los servicios almacenan datos en diferentes almacenes de datos, puede resultar difícil mantener la coherencia de datos entre estos almacenes de datos.

Motivación

Una transacción es una sola unidad de trabajo que puede implicar varios pasos, donde todos los pasos se ejecutan por completo o no se ejecuta ningún paso, lo que da como resultado un almacén de datos que conserva su estado coherente. Los términos atomicidad, coherencia, aislamiento y

durabilidad (ACID, por sus siglas en inglés) definen las propiedades de una transacción. Las bases de datos relacionales proporcionan transacciones ACID para mantener la coherencia de datos.

Para mantener la coherencia en una transacción, las bases de datos relacionales utilizan el método de confirmación en dos fases (2PC). Consiste en una fase de preparación y una fase de confirmación.

- En la fase de preparación, el proceso de coordinación solicita a los procesos participantes en la transacción (participantes) que se comprometan a confirmar o anular la transacción.
- En la fase de confirmación, el proceso de coordinación solicita a los participantes que confirmen la transacción. Si los participantes no pueden ponerse de acuerdo para realizar la confirmación en la fase de preparación, la transacción se anula.

En los sistemas distribuidos que siguen un patrón de diseño de base de datos por servicio, la confirmación en dos fases no es una opción. Esto se debe a que cada transacción se distribuye en varias bases de datos y no existe un único controlador que pueda coordinar un proceso similar a la confirmación en dos fases de los almacenes de datos relacionales. En este caso, una solución es utilizar el patrón de orquestación de la saga.

Aplicabilidad

Use el patrón de orquestación de la saga cuando:

- El sistema requiere integridad y coherencia de los datos en las transacciones distribuidas que abarcan varios almacenes de datos.
- El almacén de datos no proporciona la confirmación en dos fases para proporcionar transacciones ACID, y la implementación de la confirmación en dos fases dentro de los límites de las aplicaciones es una tarea compleja.
- Tiene bases de datos NoSQL, que no proporcionan transacciones ACID, y necesita actualizar varias tablas dentro de una sola transacción.

Problemas y consideraciones

- Complejidad: las transacciones y los reintentos de compensación agregan complejidad al código de la aplicación, lo que puede provocar una sobrecarga de mantenimiento.
- Coherencia final: el procesamiento secuencial de las transacciones locales da como resultado una coherencia final, lo que puede ser un desafío en los sistemas que requieren una gran coherencia.

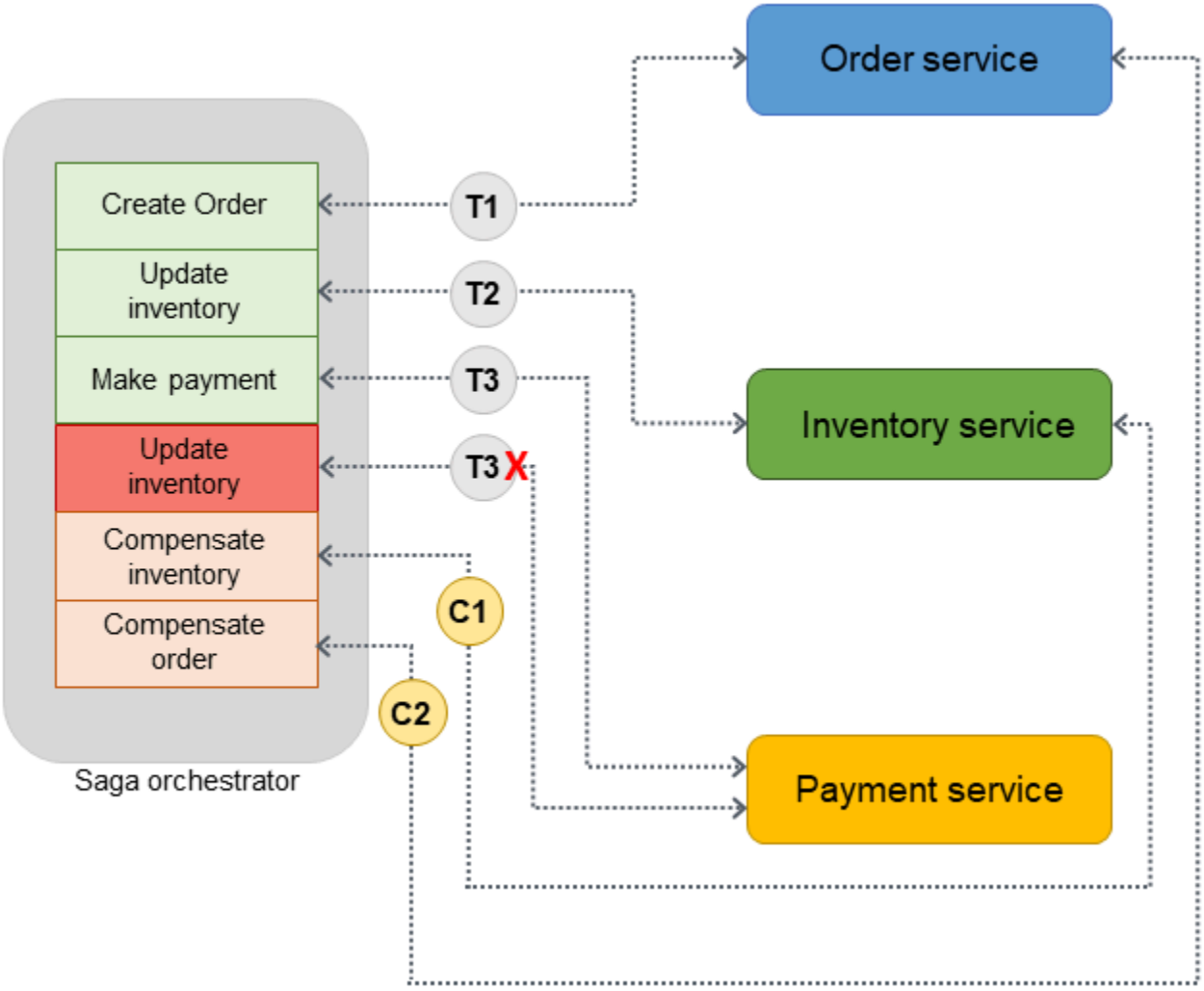
Para abordar este problema, establezca las expectativas de sus equipos empresariales con respecto al modelo de coherencia o cambie a un almacén de datos que ofrezca una coherencia sólida.

- Idempotencia: los participantes de la saga tienen que ser idempotentes para permitir la ejecución repetida en caso de que se produzcan errores transitorios provocados por bloqueos inesperados o errores del orquestador.
- Aislamiento de transacciones: la saga carece de aislamiento de transacciones. La orquestación simultánea de las transacciones puede provocar datos obsoletos. Recomendamos utilizar el bloqueo semántico para gestionar estos escenarios.
- Observabilidad: la observabilidad se refiere al registro y el seguimiento detallados para solucionar problemas en el proceso de ejecución y orquestación. Esto se vuelve importante cuando aumenta el número de participantes de la saga, lo que dificulta la depuración.
- Problemas de latencia: las transacciones compensatorias pueden agregar latencia al tiempo de respuesta total cuando la saga consta de varios pasos. Evite las llamadas sincrónicas en estos casos.
- Punto único de error: el orquestador puede convertirse en un punto único de error porque coordina toda la transacción. En algunos casos, se prefiere el patrón de coreografía de la saga debido a este problema.

Implementación

Arquitectura de alto nivel

En el siguiente diagrama de arquitectura, la orquestador de la saga tiene tres participantes: el servicio de pedidos, el servicio de inventario y el servicio de pago. Se requieren tres pasos para completar la transacción: T1, T2 y T3. El orquestador de la saga conoce los pasos y los ejecuta en el orden requerido. Cuando se produce un error en el paso T3 (falta de pago), el orquestador ejecuta las transacciones de compensación C1 y C2 para restaurar los datos al estado inicial.



Puede utilizar AWS Step Functions para implementar la orquestación de la saga cuando la transacción se distribuye en varias bases de datos.

Implementación mediante los servicios de AWS

La solución de ejemplo utiliza el flujo de trabajo estándar de Step Functions para implementar el patrón de orquestación de la saga.

Cuando un cliente llama a la API, se invoca la función de Lambda y el preprocesamiento se realiza en la función de Lambda. La función inicia el flujo de trabajo de Step Functions para empezar a procesar la transacción distribuida. Si no es necesario preprocesar, puede iniciar el flujo de trabajo de Step Functions directamente desde API Gateway sin usar la función de Lambda.

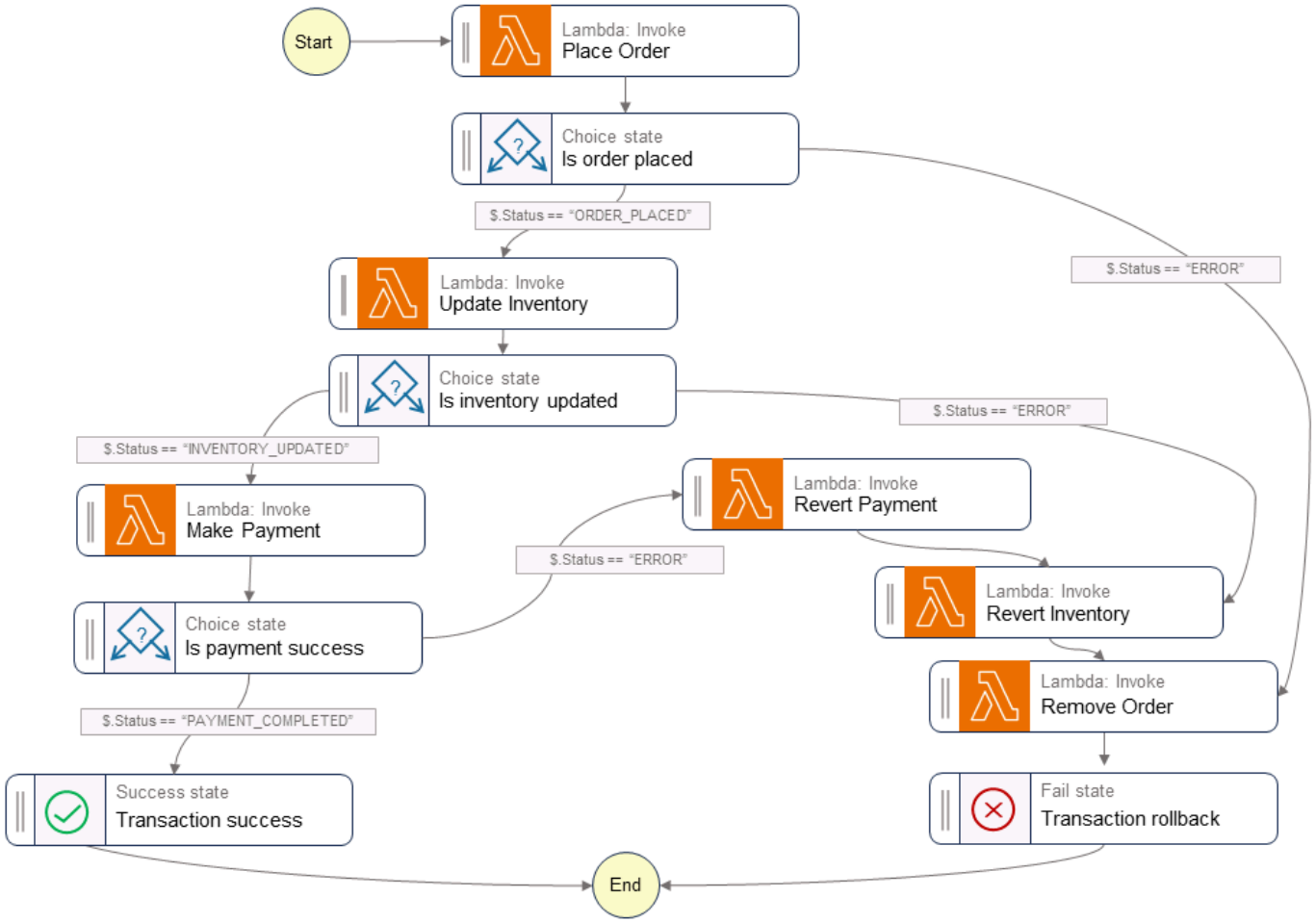
El uso de Step Functions mitiga el problema del punto único de error, que es inherente a la implementación del patrón de orquestación de la saga. Step Functions tiene una tolerancia a errores integrada y mantiene la capacidad de servicio en varias zonas de disponibilidad de cada región de AWS para proteger las aplicaciones contra los errores de máquinas individuales o del centro de datos. Esto ayuda a garantizar una alta disponibilidad tanto para el servicio en sí como para el flujo de trabajo de la aplicación que opera.

Flujo de trabajo de Step Functions

La máquina de estados de Step Functions le permite configurar los requisitos de flujo de control basados en decisiones para la implementación del patrón. El flujo de trabajo de Step Functions llama a los servicios individuales de realización de pedidos, actualización del inventario y procesamiento de pagos para completar la transacción y envía una notificación de evento para su posterior procesamiento. El flujo de trabajo de Step Functions actúa como el orquestador para coordinar las transacciones. Si el flujo de trabajo contiene algún error, el orquestador ejecuta las transacciones compensatorias para garantizar que la integridad de los datos se mantenga en todos los servicios.

En el siguiente diagrama se muestran los pasos que se ejecutan en el flujo de trabajo de Step Functions. Los pasos `Place Order`, `Update Inventory` y `Make Payment` indican la ruta correcta. Se realiza el pedido, se actualiza el inventario y se procesa el pago antes de que se devuelva el estado `Success` al intermediario.

Las funciones de Lambda `Revert Payment`, `Revert Inventory` y `Remove Order` indican las transacciones compensatorias que el orquestador ejecuta cuando se produce un error en algún paso del flujo de trabajo. Si se produce un error en el flujo de trabajo en el paso `Update Inventory`, el orquestador llama a los pasos `Revert Inventory` y `Remove Order` antes de devolver el estado `Fail` al intermediario. Estas transacciones compensatorias garantizan que se mantenga la integridad de los datos. El inventario vuelve a su nivel original y el pedido se revierte.



Código de muestra

El siguiente código de ejemplo muestra cómo puede crear un orquestador de saga mediante Step Functions. Para ver el código completo, consulte el repositorio de GitHub para ver este ejemplo.

Definiciones de tareas

```
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);
```

Definiciones de función de paso y máquina de estados

```
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
});
```

Repositorio GitHub

Para obtener una implementación completa de la arquitectura de ejemplo para este patrón, consulte el repositorio de GitHub en https://github.com/aws-samples/saga-orchestration-netcore-blog.

Referencias de blogs

- Creación de una aplicación distribuida sin servidor mediante el patrón de orquestación de la saga

Contenido relacionado

- Patrón de coreografía de la saga
- Patrón de bandeja de salida transaccional

Videos

En el video siguiente se explica cómo implementar el patrón de orquestación de la saga mediante AWS Step Functions.

Patrón de dispersión y recopilación

Intención

El patrón de dispersión y recopilación es un patrón de enrutamiento de mensajes que implica transmitir solicitudes similares o relacionadas a varios destinatarios y volver a agrupar sus respuestas en un solo mensaje mediante un componente denominado agregador. Este patrón ayuda a lograr la paralelización, reduce la latencia del procesamiento y gestiona la comunicación asincrónica. Es sencillo implementar el patrón de dispersión y recolección mediante un enfoque sincrónico, pero un enfoque más eficaz implica implementarlo como enrutamiento de mensajes en la comunicación asincrónica, con un servicio de mensajería o sin este.

Motivación

En el procesamiento de solicitudes, una solicitud que puede tardar mucho tiempo en procesarse de manera secuencial se puede dividir en varias solicitudes que se procesan en paralelo. También puede enviar solicitudes a varios sistemas externos mediante llamadas a la API para obtener una respuesta. El patrón de dispersión y recopilación es útil cuando se necesita información de varios orígenes. La dispersión y recopilación agregan los resultados para ayudar a tomar una decisión informada o a seleccionar la mejor respuesta para la solicitud.

El patrón de dispersión y recopilación consta de dos fases, como su nombre lo indica:

- La fase de dispersión procesa el mensaje de la solicitud y lo envía a varios destinatarios en paralelo. Durante esta fase, la aplicación dispersa las solicitudes por la red y continúa ejecutándose sin esperar respuestas inmediatas.
- Durante la fase de recopilación, la aplicación recopila las respuestas de los destinatarios y las filtra o combina en una respuesta unificada. Una vez recopiladas todas las respuestas, se pueden agregar en una sola respuesta o se puede elegir la mejor para su procesamiento posterior.

Aplicabilidad

Utilice el patrón de dispersión y recopilación cuando suceda lo siguiente:

- Planea agregar y consolidar los datos de varias API para crear una respuesta precisa. El patrón consolida la información de orígenes dispares en un todo cohesivo. Por ejemplo, un sistema de

reservas puede hacer una solicitud a varios destinatarios para obtener cotizaciones de varios socios externos.

- La misma solicitud se debe enviar a varios destinatarios de manera simultánea para completar una transacción. Así, por ejemplo, puede utilizar este patrón para consultar los datos del inventario en paralelo para verificar la disponibilidad de un producto.
- Desea implementar un sistema fiable y escalable en el que se pueda lograr el equilibrio de cargas mediante la distribución de las solicitudes entre varios destinatarios. Si se produce un error en un destinatario o este tiene una carga alta, los demás destinatarios pueden seguir procesando las solicitudes.
- Desea optimizar el rendimiento al implementar consultas complejas que impliquen varios orígenes de datos. Puede dispersar la consulta en las bases de datos correspondientes, recopilar los resultados parciales y combinarlos en una respuesta completa.
- Está implementando un tipo de procesamiento map-reduce en el que la solicitud de datos se enruta a varios puntos de conexión de procesamiento de datos para su partición y replicación. Los resultados parciales se filtran y combinan para crear la respuesta correcta.
- Desea distribuir las operaciones de escritura en un espacio de claves de partición en las cargas de trabajo de escritura intensiva en bases de datos de clave-valor. El agregador lee los resultados: Para ello, consulta los datos de cada partición y, a continuación, los consolida en una sola respuesta.

Problemas y consideraciones

- Tolerancia a errores: este patrón se basa en varios receptores que funcionan en paralelo, por lo que es esencial gestionar los errores de manera adecuada. Para mitigar el impacto de los errores de los destinatarios en todo el sistema, puede implementar estrategias como la redundancia, la replicación y la detección de errores.
- Límites de escalado horizontal: a medida que aumenta el número total de nodos de procesamiento, también aumenta la sobrecarga de red asociada. Cada solicitud que implique la comunicación a través de la red puede aumentar la latencia y afectar de manera negativa a las ventajas de la paralelización.
- Obstáculos en el tiempo de respuesta: en el caso de las operaciones para las que es necesario que se procesen todos los destinatarios antes de completar el procesamiento final, el rendimiento del sistema en general se ve limitado por el tiempo de respuesta del destinatario más lento.

- Respuestas parciales: cuando las solicitudes se distribuyen entre varios destinatarios, es posible que se agote el tiempo de espera de algunos destinatarios. En estos casos, la implementación debe comunicar al cliente que la respuesta está incompleta. También puede mostrar los detalles de agregación de respuestas mediante una interfaz de usuario de frontend.
- Coherencia de datos: cuando procese datos de varios destinatarios, debe considerar con detenimiento las técnicas de sincronización de datos y resolución de conflictos para garantizar que los resultados agregados finales sean precisos y coherentes.

Implementación

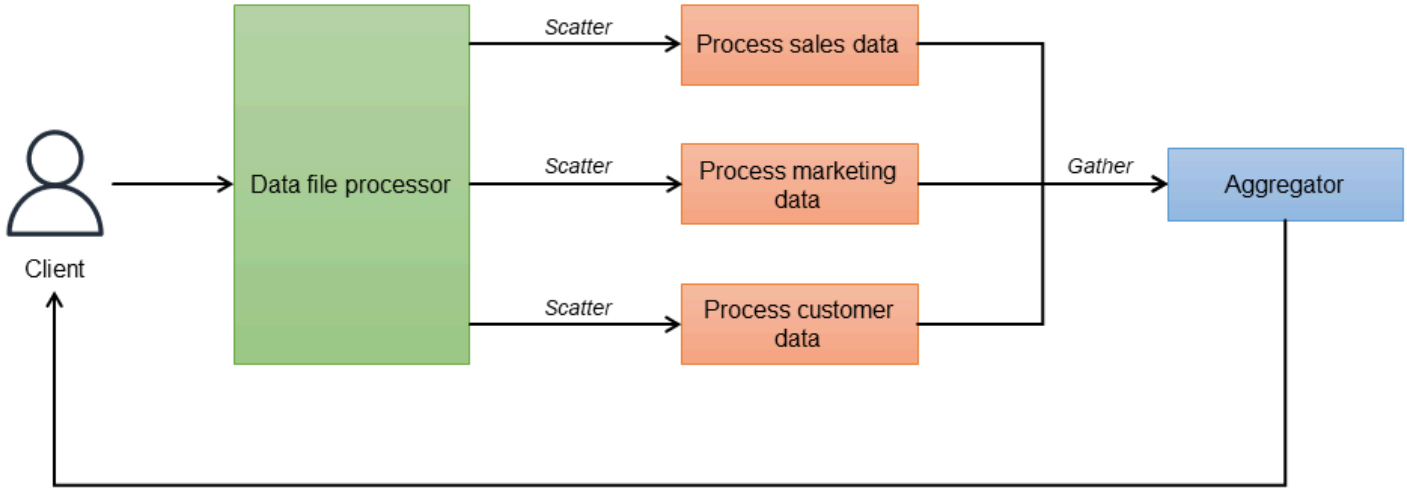
Arquitectura de alto nivel

El patrón de dispersión y recopilación utiliza un controlador raíz para distribuir las solicitudes a los destinatarios que las procesarán. Durante la fase de dispersión, este patrón puede utilizar dos mecanismos para enviar mensajes a los destinatarios:

- Dispersión por distribución: la aplicación tiene una lista conocida de destinatarios a los que hay que llamar para obtener los resultados. Los destinatarios pueden ser procesos distintos que tienen funciones únicas o un solo proceso que se escaló horizontalmente para distribuir la carga de procesamiento. Si alguno de los nodos de procesamiento agota el tiempo de espera o presenta demoras en la respuesta, el controlador puede redistribuir el procesamiento a otro nodo.
- Dispersión por subasta: la aplicación difunde el mensaje a los destinatarios interesados mediante un patrón de publicación y suscripción. En este caso, los destinatarios pueden suscribirse al mensaje o anular la suscripción en cualquier momento.

Dispersión por distribución

En el método de dispersión por distribución, el controlador raíz divide la solicitud entrante en tareas independientes y las asigna a los destinatarios disponibles (la fase de dispersión). Cada destinatario (proceso, contenedor o función de Lambda) trabaja de manera independiente y paralela en su computación y produce una parte de la respuesta. Cuando los destinatarios finalizan sus tareas, envían sus respuestas a un agregador (la fase de recopilación). El agregador combina las respuestas parciales y devuelve el resultado final al cliente. En el diagrama siguiente se ilustra este flujo de trabajo.



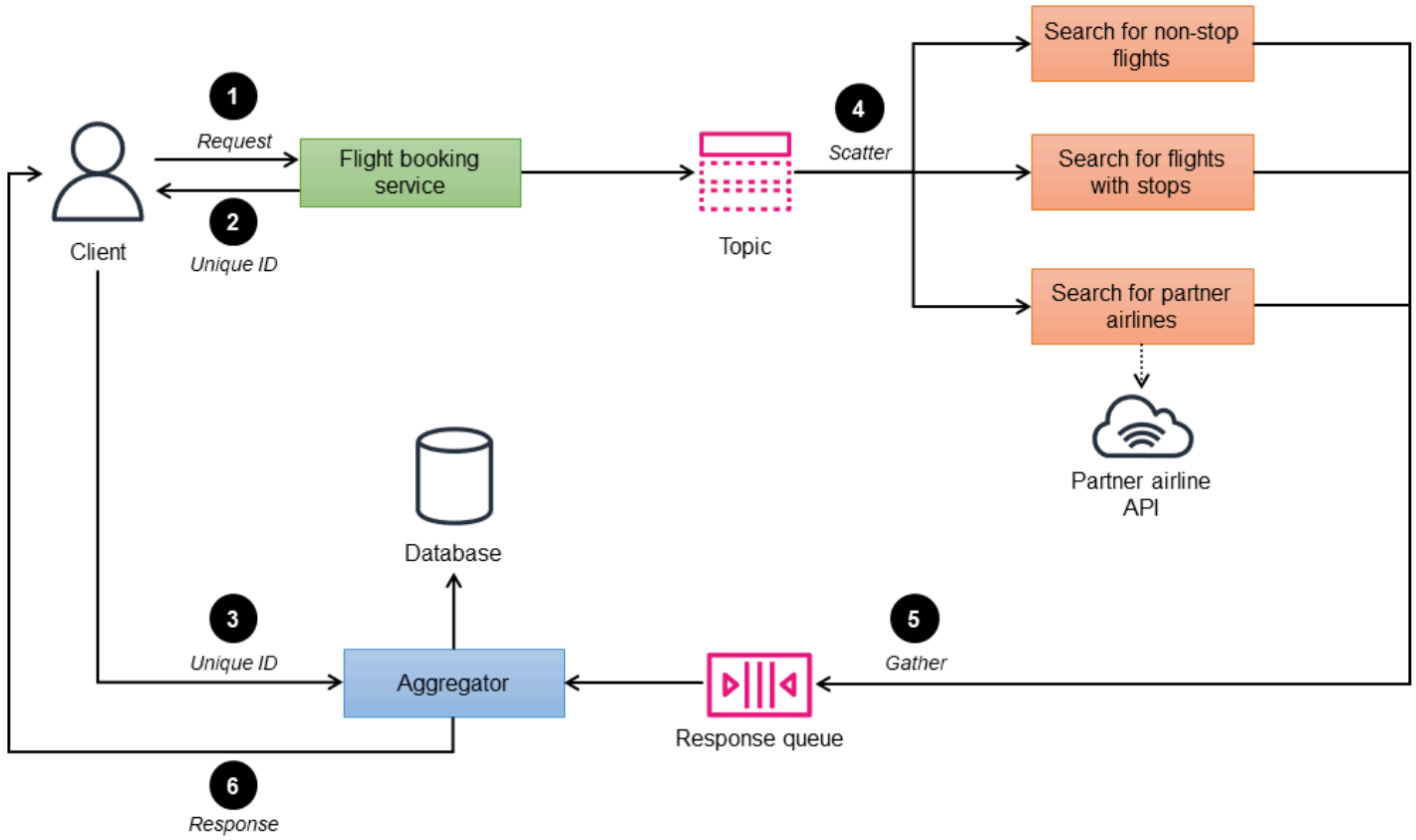
El controlador (procesador de archivos de datos) orquesta todo el conjunto de invocaciones y conoce todos los puntos de conexión de reserva a los que se debe llamar. Este puede configurar un parámetro de tiempo de espera para ignorar las respuestas que tardan demasiado. Una vez enviadas las solicitudes, el agregador espera a que lleguen las respuestas de cada punto de conexión. Para implementar la resiliencia, cada microservicio se puede implementar con varias instancias para equilibrar la carga. El agregador obtiene los resultados, los combina en un solo mensaje de respuesta y elimina los datos duplicados antes de continuar con su procesamiento. Se ignoran las respuestas que superan el tiempo de espera. El controlador también puede actuar como un agregador en lugar de utilizar un servicio de agregación independiente.

Dispersión por subasta

Si el controlador no sabe cuáles son los destinatarios o si los destinatarios están acoplados de manera débil, puede utilizar el método de dispersión por subasta. En este método, los destinatarios se suscriben a un tema y el controlador publica la solicitud en dicho tema. Los destinatarios publican los resultados en una cola de respuestas. Como el controlador raíz no conoce a los destinatarios, el proceso de recopilación utiliza un agregador (otro patrón de mensajería) para recopilar las respuestas y resumirlas en un solo mensaje de respuesta. El agregador utiliza un id. exclusivo para identificar un grupo de solicitudes.

Así, por ejemplo, en el diagrama siguiente, el método de dispersión por subasta se utiliza para implementar un servicio de reserva de vuelos para el sitio web de una aerolínea. El sitio web permite a los usuarios buscar y mostrar los vuelos de la propia aerolínea y de las empresas asociadas. Además, debe mostrar el estado de la búsqueda en tiempo real. El servicio de reservas de vuelos

consta de tres microservicios de búsqueda: vuelos sin escalas, vuelos con escalas y aerolíneas asociadas. La búsqueda de aerolíneas asociadas llama a los puntos de conexión de la API del socio para obtener las respuestas.

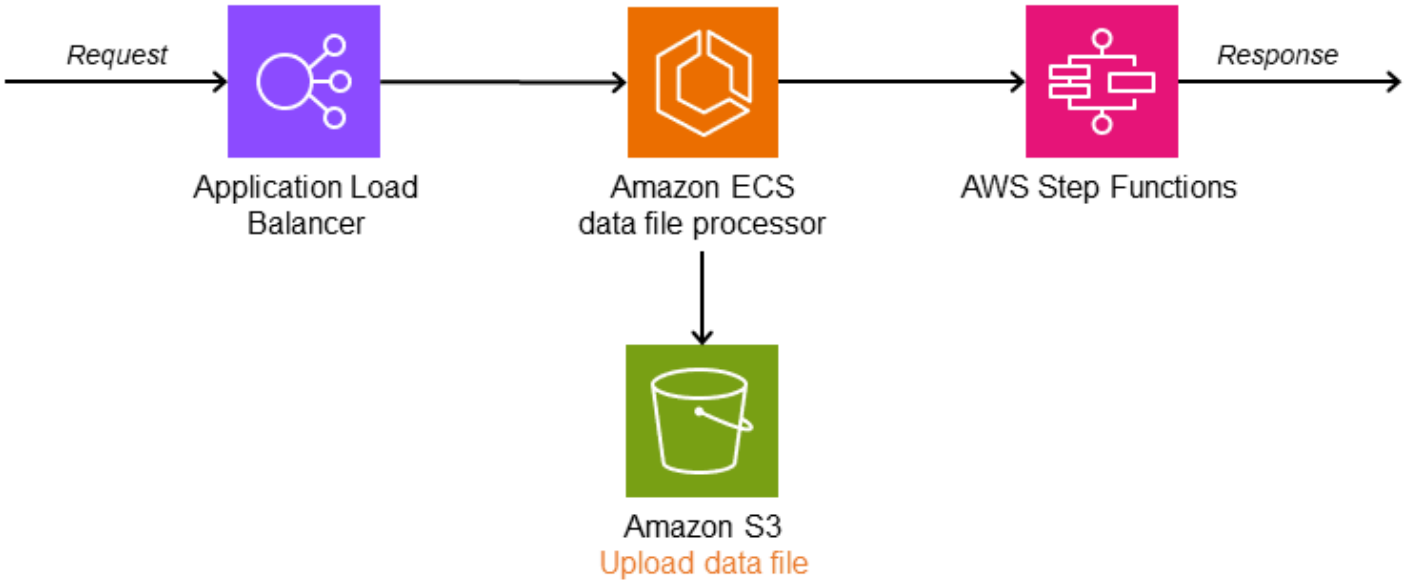


1. El servicio de reserva de vuelos (controlador) toma los criterios de búsqueda como datos del cliente y procesa y publica la solicitud relacionada con el tema.
2. El controlador utiliza un id. exclusivo para identificar cada grupo de solicitudes.
3. El cliente envía el id. exclusivo al agregador para el paso 6.
4. Los microservicios de búsqueda de reservas que se suscribieron al tema de reserva reciben la solicitud.
5. Los microservicios procesan la solicitud y devuelven la disponibilidad de asientos a una cola de respuesta según los criterios de búsqueda indicados.
6. El agregador recopila todos los mensajes de respuesta almacenados en una base de datos temporal, agrupa los vuelos por un id. exclusivo, crea una sola respuesta unificada y la devuelve al cliente.

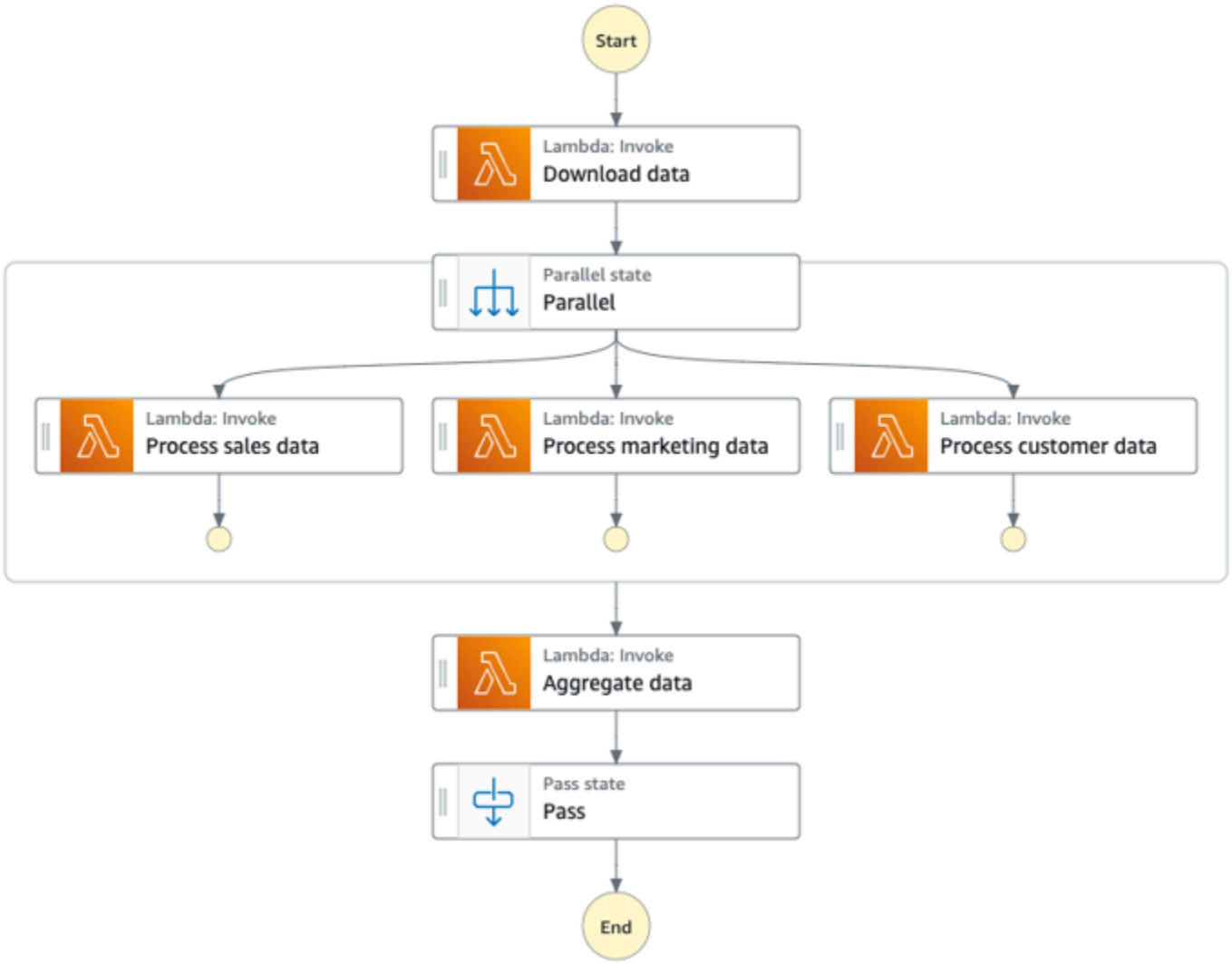
Implementación mediante Servicios de AWS

Dispersión por distribución

En la arquitectura siguiente, el controlador raíz es un procesador de archivos de datos (Amazon ECS) que divide los datos de las solicitudes entrantes en buckets individuales de Amazon Simple Storage Service (Amazon S3) e inicia un flujo de trabajo de AWS Step Functions. El flujo de trabajo descarga los datos e inicia el procesamiento de los archivos en paralelo. El estado `Parallel` espera a que todas las tareas devuelvan una respuesta. Una función de AWS Lambda agrega los datos y los guarda en Amazon S3.

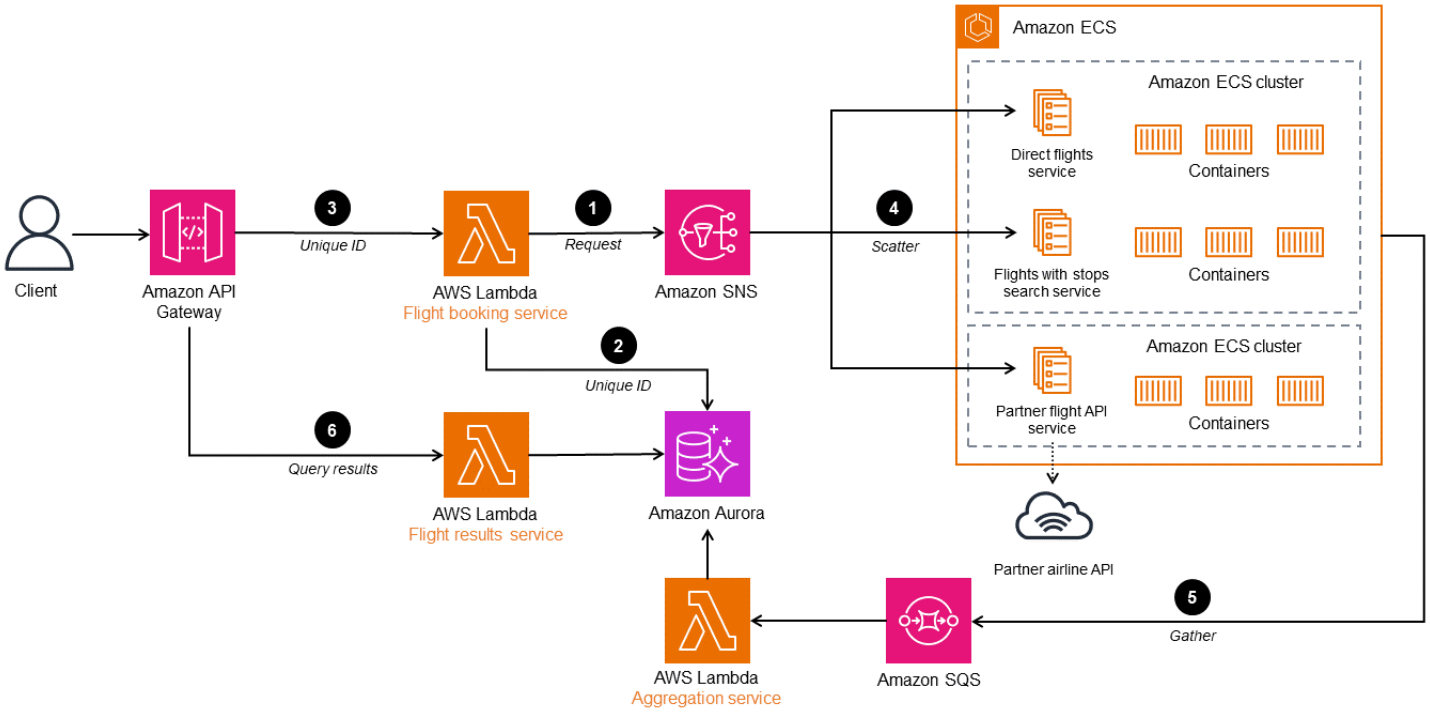


En el diagrama siguiente se ilustra el flujo de trabajo de Step Functions con el estado `Parallel`.



Dispersión por subasta

En el diagrama siguiente se muestra una arquitectura de AWS para el método de dispersión por subasta. El servicio de reserva de vuelos del controlador raíz dispersa la solicitud de búsqueda de vuelos en varios microservicios. El canal de publicación y suscripción se implementa con Amazon Simple Notification Service (Amazon SNS), que es un servicio de mensajería administrado para las comunicaciones. Amazon SNS admite mensajes entre aplicaciones de microservicios desacopladas o comunicaciones directas con los usuarios. Puede implementar los microservicios del destinatario en Amazon Elastic Kubernetes Service (Amazon EKS) o Amazon Elastic Container Service (Amazon ECS) para mejorar la administración y escalabilidad. El servicio de resultados de vuelos devuelve los resultados al cliente. Se puede implementar en AWS Lambda u otros servicios de orquestación de contenedores, como Amazon ECS o Amazon EKS.



1. El servicio de reserva de vuelos (controlador) toma los criterios de búsqueda como datos del cliente y procesa y publica la solicitud relacionada con el tema de SNS.
2. El controlador publica el id. exclusivo en una base de datos de Amazon Aurora para identificar la solicitud.
3. El cliente envía el id. exclusivo al cliente para el paso 6.
4. Los microservicios de búsqueda de reservas que se suscribieron al tema de reserva reciben la solicitud.
5. Los microservicios procesan la solicitud y devuelven la disponibilidad de asientos a una cola de respuestas en Amazon Simple Queue Service (Amazon SQS) según los criterios de búsqueda indicados. El agregador recopila todos los mensajes de respuesta y los almacena en una base de datos temporal.
6. El servicio de resultados de vuelos agrupa los vuelos por un id. exclusivo, crea una sola respuesta unificada y la envía al cliente.

Si quiere agregar otra búsqueda de aerolíneas a esta arquitectura, agregue un microservicio que se suscriba al tema de SNS y lo publique en la cola de SQS.

En resumen, el patrón de dispersión y recopilación permite a los sistemas distribuidos lograr una paralelización eficiente, reducir la latencia y gestionar sin problemas la comunicación asincrónica.

Repositorio GitHub

Para obtener una implementación completa de la arquitectura de ejemplo para este patrón, consulte el repositorio de GitHub en https://github.com/aws-samples/asynchronous-messaging-workshop/tree/master/code/lab-3.

Taller

- Scatter-gather lab en el taller Decoupled Microservices

Referencias de blogs

- Application integration patterns for microservices

Contenido relacionado

- Publish-subscribe pattern

Patrón de higo estrangulador

Intención

El patrón de higo estrangulador ayuda a migrar una aplicación monolítica a una arquitectura de microservicios de manera gradual, con un riesgo menor de transformación y una disrupción menor empresarial.

Motivación

Las aplicaciones monolíticas se desarrollan para proporcionar la mayor parte de sus funciones en un solo proceso o contenedor. El código está acoplado de manera ajustada. Por este motivo, para hacer cambios en las aplicaciones son necesarias nuevas pruebas exhaustivas para evitar problemas de regresión. Los cambios no se pueden probar de manera aislada, lo que afecta a la duración del ciclo. A medida que la aplicación se enriquece con más características, la alta complejidad puede implicar que se dedique más tiempo al mantenimiento, un aumento del tiempo de comercialización y, en consecuencia, una ralentización de la innovación de los productos.

Cuando el tamaño de la aplicación se escala, aumenta la carga cognitiva del equipo y puede provocar que los límites de responsabilidad del equipo no estén claros. No es posible escalar las características individuales según la carga. Es necesario escalar toda la aplicación para permitir la carga máxima. A medida que los sistemas envejecen, la tecnología puede quedar obsoleta, lo que aumenta los costos de soporte. Las aplicaciones antiguas y monolíticas siguen las prácticas recomendadas que estaban disponibles en el momento del desarrollo y no se diseñaron para distribuirse.

Cuando se migra una aplicación monolítica a una arquitectura de microservicios, se puede dividir en componentes más pequeños. Estos componentes se pueden escalar de manera independiente, se pueden lanzar de manera independiente y pueden ser responsabilidad de equipos individuales. Esto da lugar a una mayor velocidad de cambio, ya que los cambios se localizan y se pueden probar y publicar de manera rápida. Los cambios tienen un ámbito de impacto menor porque los componentes están acoplados de manera débil y se pueden implementar de manera individual.

Sustituir por completo un monolito por una aplicación de microservicios mediante la reescritura o la refactorización del código es una tarea enorme y un riesgo grande. Una migración a gran escala, en la que el monolito se migra en una sola operación, supone un riesgo de transformación y una

disrupción empresarial. Mientras se refactoriza la aplicación, es extremadamente difícil o incluso imposible agregar características nuevas.

Una manera de resolver este problema es utilizar el patrón de higo estrangulador, que introdujo Martin Fowler. Este patrón implica pasar a los microservicios mediante la extracción gradual de características y la creación de una aplicación nueva en torno al sistema existente. Las características del monolito se sustituyen de manera gradual por microservicios. Los usuarios de las aplicaciones pueden utilizar las características recién migradas de manera progresiva. Cuando se trasladen todas las características al sistema nuevo, la aplicación monolítica se podrá retirar de manera segura.

Aplicabilidad

Utilice el patrón de higo estrangulador cuando suceda lo siguiente:

- Desea migrar la aplicación monolítica de manera gradual a una arquitectura de microservicios.
- Un enfoque de migración radical es riesgoso debido al tamaño y la complejidad del monolito.
- La empresa quiere agregar características nuevas y no puede esperar a que se complete la transformación.
- Los usuarios finales deben verse afectados lo menos posible durante la transformación.

Problemas y consideraciones

- Acceso a la base del código: para implementar el patrón de higo estrangulador, debe tener acceso a la base del código de la aplicación monolítica. A medida que se vayan migrando las características del monolito, será necesario hacer cambios pequeños en el código e implementar una capa anticorrupción en el monolito para enrutar las llamadas a microservicios nuevos. No puede interceptar llamadas sin acceso a la base del código. El acceso a la base del código también es fundamental para redirigir las solicitudes entrantes. Es posible que sea necesario refactorizar el código para que la capa del proxy pueda interceptar las llamadas de las características migradas y enrutarlas a los microservicios.
- Dominio poco claro: la descomposición prematura de los sistemas puede resultar costosa, sobre todo cuando el dominio no está claro y es posible definir mal los límites de los servicios. El diseño basado en dominios (DDD) es un mecanismo para comprender el dominio. La tormenta de eventos es una técnica para determinar los límites del dominio.

- Identificación de microservicios: puede utilizar el DDD como una herramienta clave para identificar los microservicios. Para identificar los microservicios, busque las divisiones naturales entre las clases de servicios. Numerosos servicios tendrán su propio objeto de acceso a los datos y se desacoplarán con facilidad. Los servicios que tienen una lógica empresarial relacionada y las clases que no tienen dependencias o que tienen pocas dependencias son buenos candidatos para convertirse en microservicios. Puede refactorizar el código antes de descomponer el monolito para evitar un acoplamiento ajustado. También debe tener en cuenta los requisitos de cumplimiento, la frecuencia de publicación, la ubicación geográfica de los equipos, las necesidades de escala, las necesidades tecnológicas basadas en casos de uso y la carga cognitiva de los equipos.
- Capa anticorrupción: durante el proceso de migración, cuando las características del monolito tengan que llamar a las características que se migraron como microservicios, se debe implementar una capa anticorrupción (ACL) que dirija cada llamada al microservicio correspondiente. Con el fin de desacoplar a los autores de las llamadas existentes en el monolito y evitar que se produzcan cambios en estos, la ACL funciona como un adaptador o una fachada que convierte las llamadas en una interfaz más nueva. Esto se analiza en detalle en una sección anterior sobre la implementación del patrón de ACL que figura en esta guía.
- Error en la capa del proxy: durante la migración, una capa proxy intercepta las solicitudes que van a la aplicación monolítica y las dirige al sistema heredado o al sistema nuevo. Sin embargo, esta capa del proxy puede convertirse en un solo punto de error o en un obstáculo para el rendimiento.
- Complejidad de la aplicación: los monolitos grandes son los que más se benefician del diseño de higo estrangulador. En el caso de las aplicaciones pequeñas, en las que la complejidad de una refactorización completa es baja, podría resultar más eficiente volver a escribir la aplicación en una arquitectura de microservicios en lugar de migrarla.
- Interacciones de servicios: los microservicios se pueden comunicar de manera sincrónica o asíncrona. Cuando se es necesaria una comunicación sincrónica, considere si los tiempos de espera pueden provocar el consumo de conexiones o del grupo de subprocesos y provocar que las aplicaciones tengan problemas de rendimiento. En esos casos, utilice el patrón de disyuntores para devolver el error inmediato en el caso de las operaciones en las que se puedan producir errores durante periodos prolongados. La comunicación asíncrona se puede lograr mediante el uso de eventos y colas de mensajes.
- Agregación de datos: en una arquitectura de microservicios, los datos se distribuyen en las bases de datos. Cuando sea necesario agregar datos, puede utilizar AWS AppSync en el frontend o el patrón de división de responsabilidades de consulta de comandos (CQRS) en el backend.
- Coherencia de datos: los microservicios tienen su propio almacén de datos y la aplicación monolítica también puede utilizar estos datos. Para permitir el uso compartido, puede sincronizar

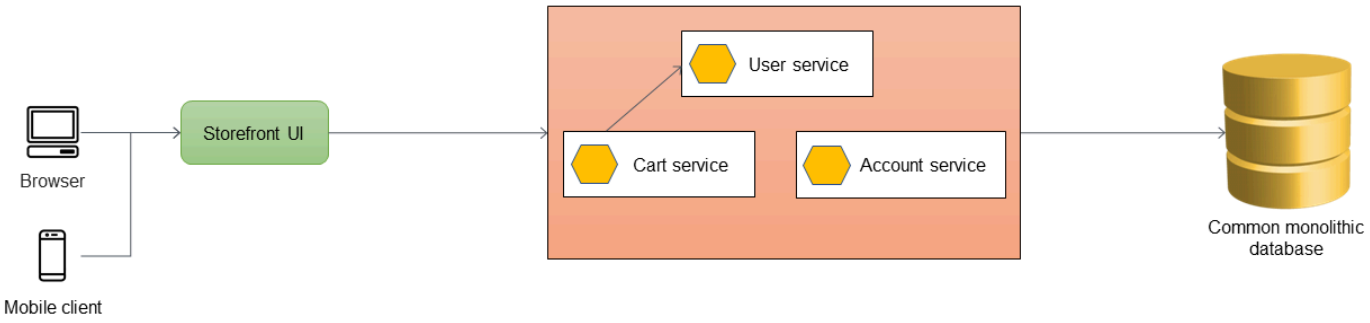
el almacén de datos de los microservicios nuevos con la base de datos de la aplicación monolítica mediante una cola y un agente. Sin embargo, esto puede provocar redundancia de datos y una coherencia final entre dos almacenes de datos, por lo que le recomendamos que lo trate como una solución táctica hasta que pueda establecer una solución a largo plazo, como un lago de datos.

Implementación

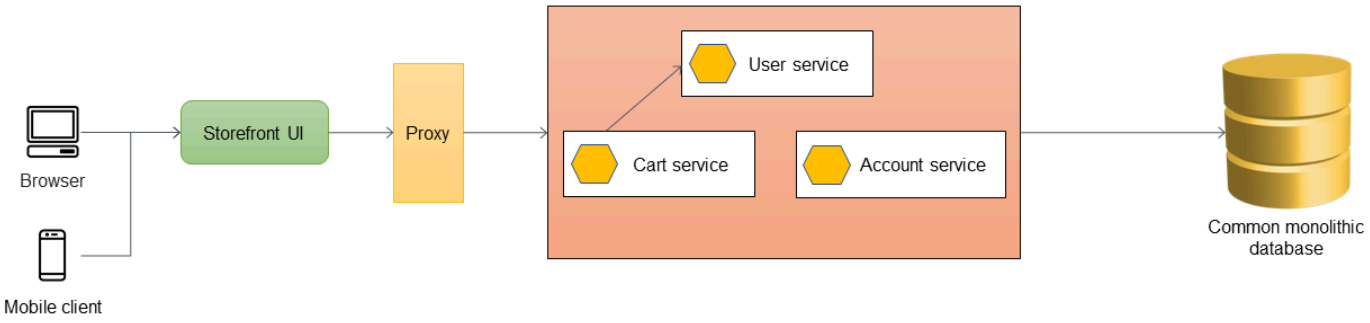
Según el patrón del higo estrangulador, se sustituye una funcionalidad específica por un servicio o una aplicación nuevos, un componente a la vez. Una capa del proxy intercepta las solicitudes que van a la aplicación monolítica y las dirige al sistema heredado o al sistema nuevo. Como la capa del proxy dirige a los usuarios a la aplicación correcta, puede agregar características al sistema nuevo y, al mismo tiempo, garantizar que siga funcionando el monolito. Con el tiempo, el sistema nuevo sustituirá todas las características del sistema anterior y podrá retirarlo del servicio.

Arquitectura de alto nivel

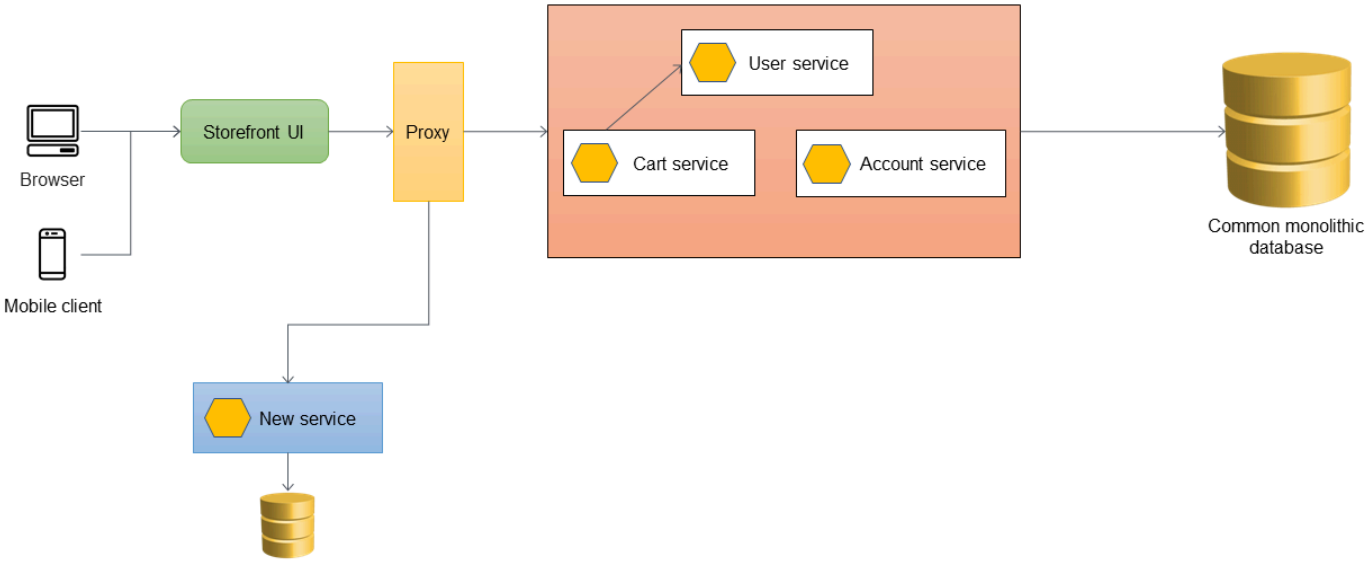
En el diagrama siguiente, una aplicación monolítica tiene tres servicios: servicio de usuario, servicio de carrito y servicio de cuentas. El servicio de carrito depende del servicio de usuario y la aplicación utiliza una base de datos relacional monolítica.



El primer paso es agregar una capa del proxy entre la interfaz de usuario de la tienda y la aplicación monolítica. Al principio, el proxy enruta todo el tráfico a la aplicación monolítica.

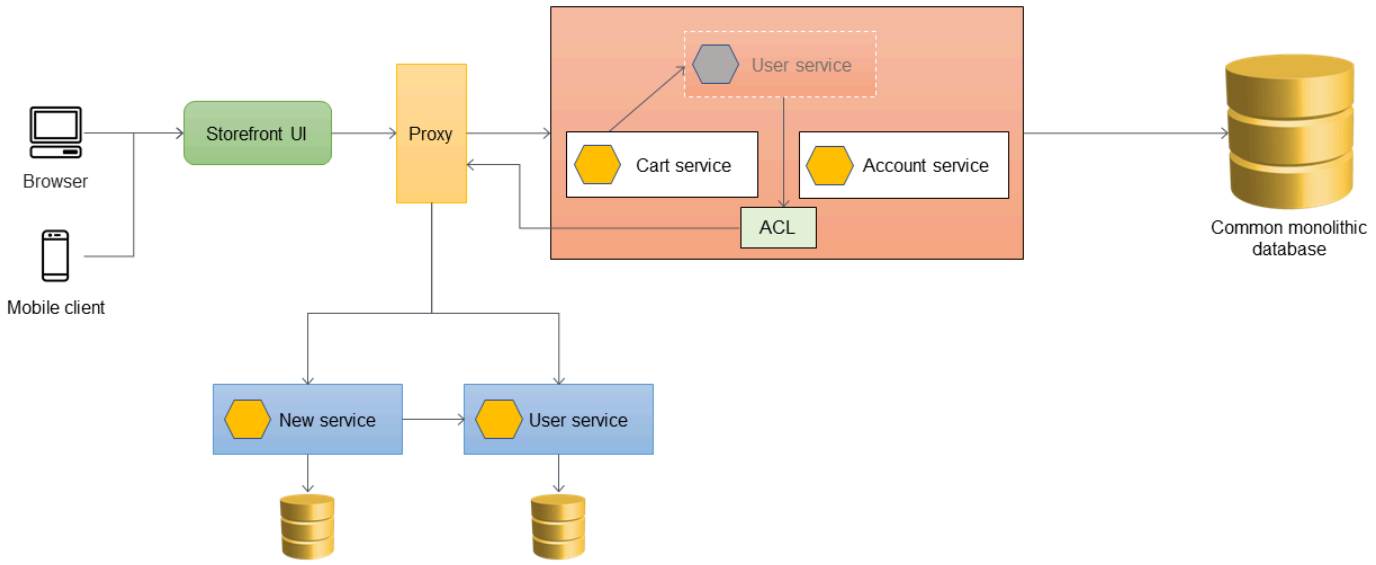


Cuando quiera agregar características nuevas a la aplicación, debe implementarlas como microservicios nuevos en lugar de agregar características al monolito existente. Sin embargo, sigue corrigiendo errores en el monolito para garantizar la estabilidad de la aplicación. En el diagrama siguiente, la capa del proxy enruta las llamadas al monolito o al microservicio nuevo según la URL de la API.



Introducción de una capa anticorrupción

En la arquitectura siguiente, el servicio de usuario se migró a un microservicio. El servicio de carrito llama al servicio de usuario, pero la implementación deja de estar disponible en el monolito. Además, es posible que la interfaz del servicio recién migrado no coincida con la interfaz anterior de la aplicación monolítica. Para abordar estos cambios, debe implementar una ACL. Durante el proceso de migración, cuando las características del monolito tienen que llamar a las características que se migraron como microservicios, la ACL convierte las llamadas a la interfaz nueva y las enruta al microservicio correspondiente.

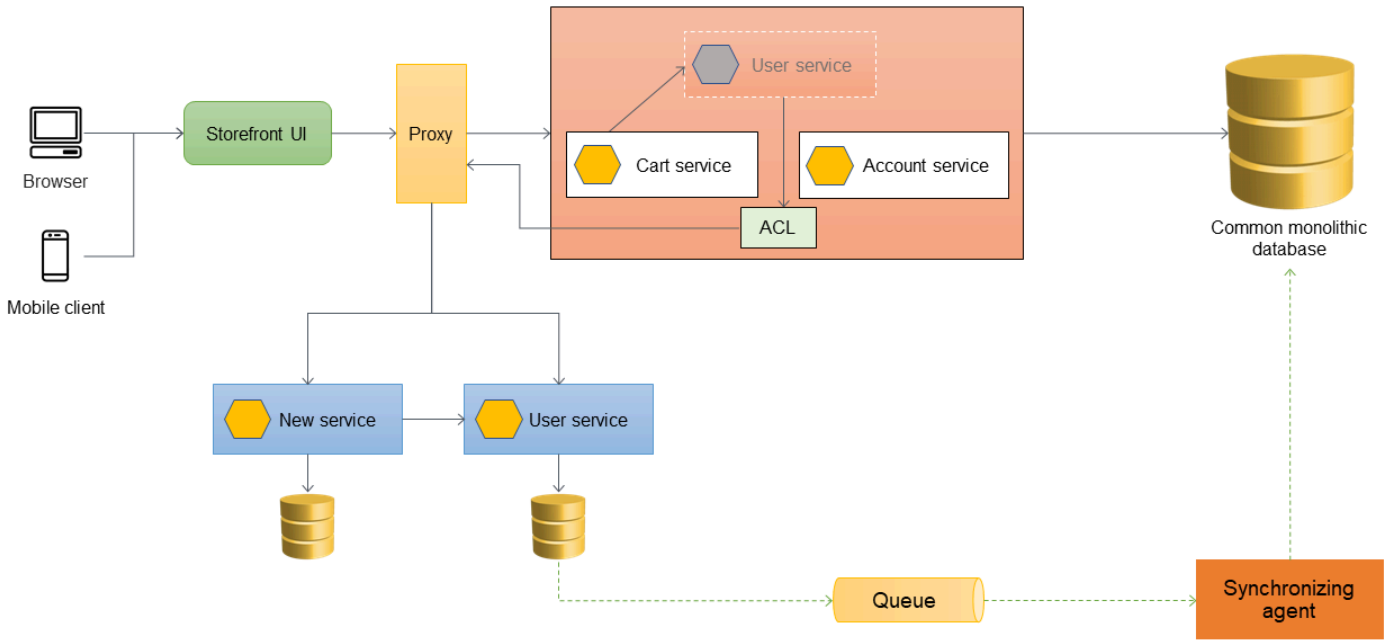


Puede implementar la ACL dentro de la aplicación monolítica como una clase específica del servicio que se migró. Por ejemplo, `UserServiceFacade` o `UserServiceAdapter`. La ACL debe retirarse una vez que todos los servicios dependientes se hayan migrado a la arquitectura de microservicios.

Cuando se utiliza la ACL, el servicio de carrito sigue llamando al servicio de usuario en el monolito y el servicio de usuario redirige la llamada al microservicio a través de la ACL. El servicio de carrito debe seguir llamando al servicio de usuario sin saber de la migración del microservicio. Este acoplamiento débil es necesario para reducir la regresión y la disrupción empresarial.

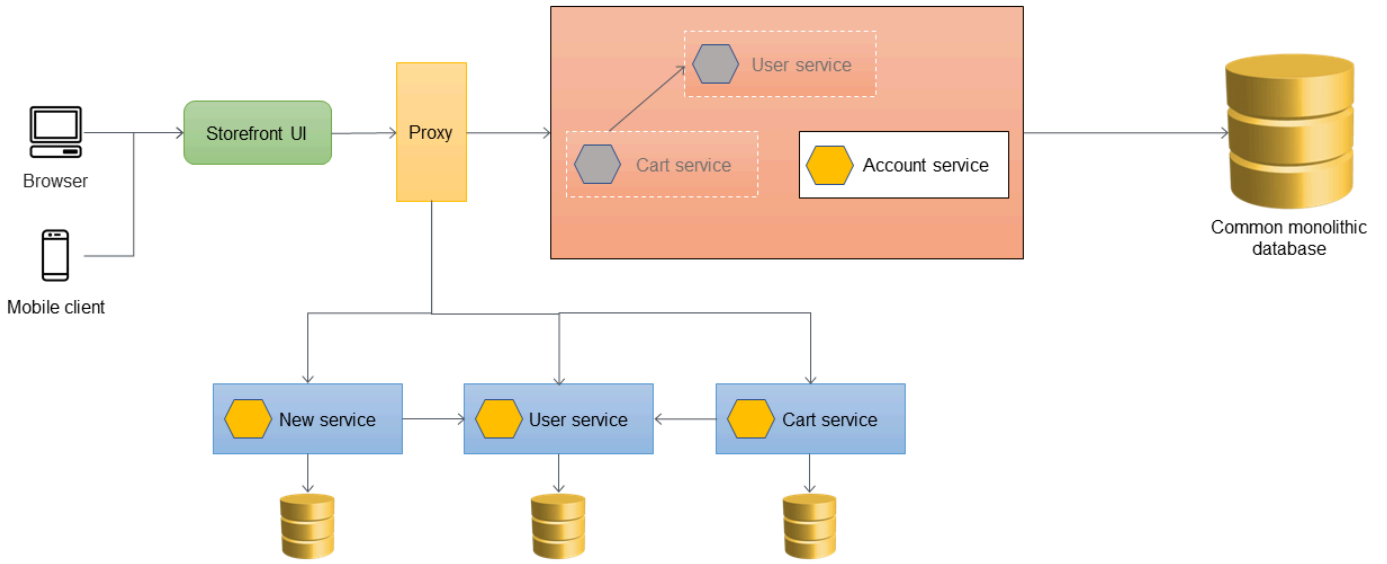
Gestión de la sincronización de los datos

Se recomienda que el microservicio debe ser responsable de sus datos. El servicio de usuario almacena sus datos en su propio almacén de datos. Es posible que tenga que sincronizar los datos con la base de datos monolítica para gestionar las dependencias, como la generación de informes, y para que las aplicaciones posteriores que aún no están preparadas puedan acceder de manera directa a los microservicios. Es posible que la aplicación monolítica también requiera los datos para otras funciones y componentes que aún no se migraron a los microservicios. Por lo tanto, es necesaria la sincronización de datos entre el microservicio nuevo y el monolito. Para sincronizar los datos, puede introducir un agente de sincronización entre el microservicio del usuario y la base de datos monolítica, como se muestra en el diagrama siguiente. El microservicio de usuario envía un evento a la cola cada vez que se actualiza su base de datos. El agente de sincronización escucha la cola y actualiza de manera continua la base de datos monolítica. Por último, los datos de la base de datos monolítica son coherentes con los datos que se sincronizan.



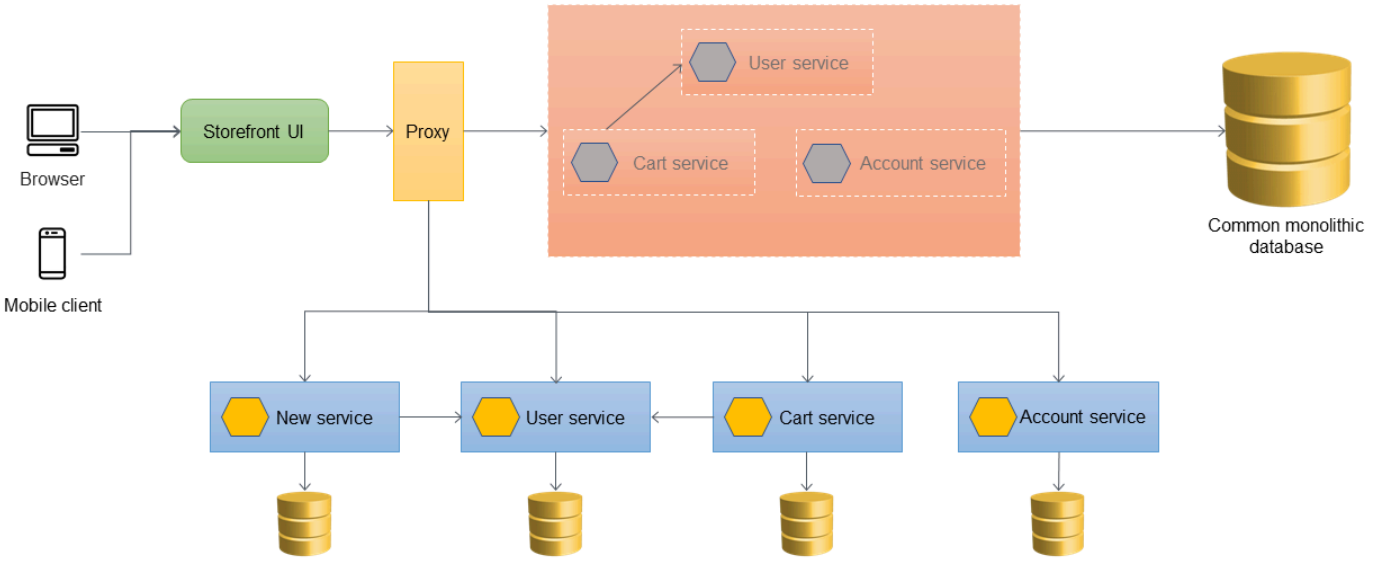
Migración de servicios adicionales

Cuando el servicio de carrito se migra de la aplicación monolítica, su código se revisa para llamar de manera directa al servicio nuevo, de modo que la ACL ya no enrute esas llamadas. En el siguiente diagrama se ilustra esta arquitectura.

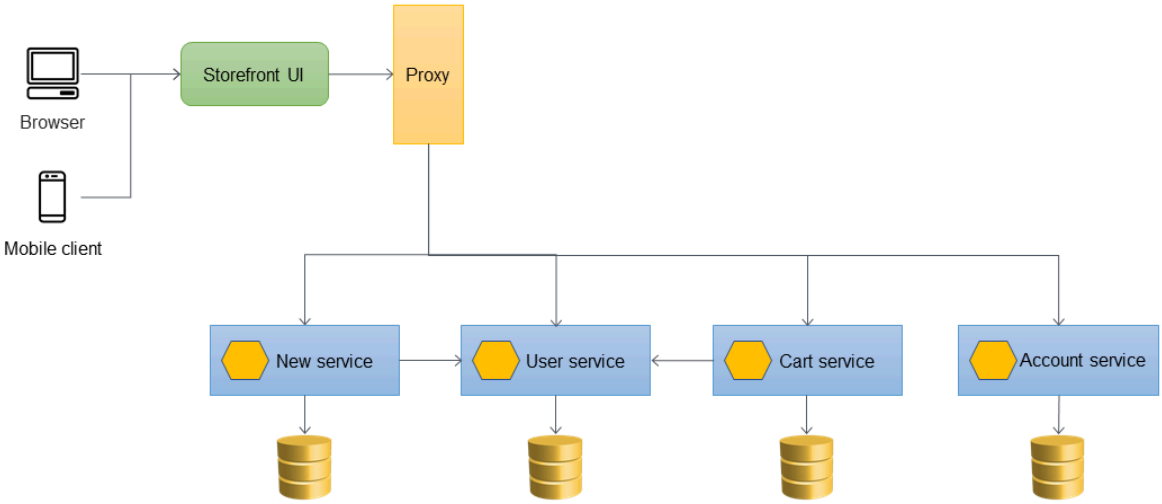


En el diagrama siguiente se muestra el estado final de estrangulamiento, en el que todos los servicios se migraron del monolito y solo queda el esqueleto del monolito. Los datos históricos se

pueden migrar a los almacenes de datos que son responsabilidad de los servicios individuales. La ACL se puede eliminar y el monolito está listo para su retirada en esta etapa.



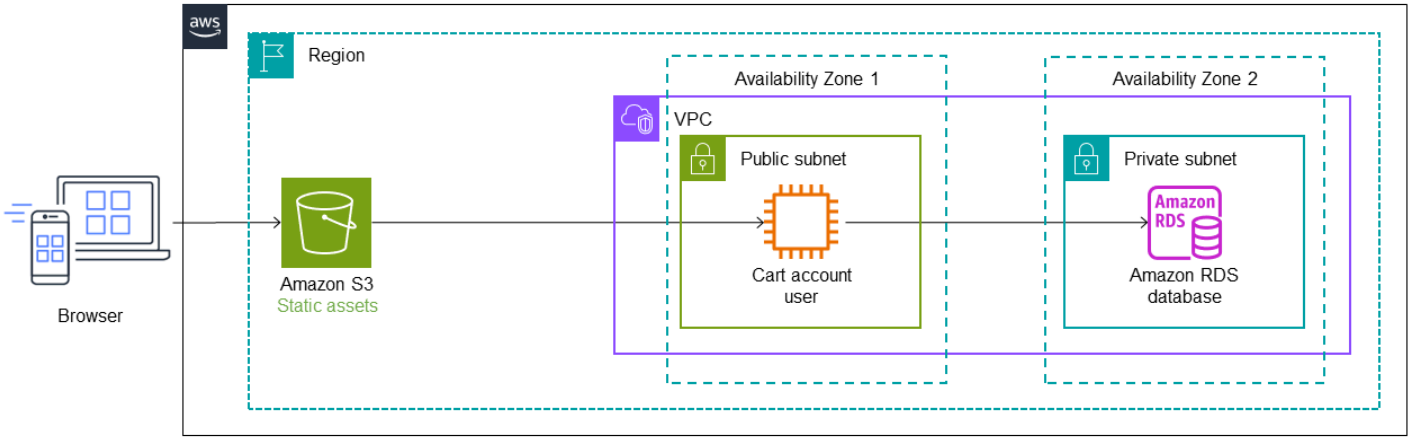
En el diagrama siguiente se muestra la arquitectura final tras la retirada de la aplicación monolítica. Puede alojar los microservicios individuales a través de una URL basada en recursos (por ejemplo `http://www.storefront.com/user`) o a través de su propio dominio (por ejemplo `http://user.storefront.com`), según los requisitos de la aplicación. Para más información sobre los métodos principales para exponer las API HTTP a los consumidores principales mediante nombres de host y rutas, consulte la sección Patrones de enrutamiento de API.



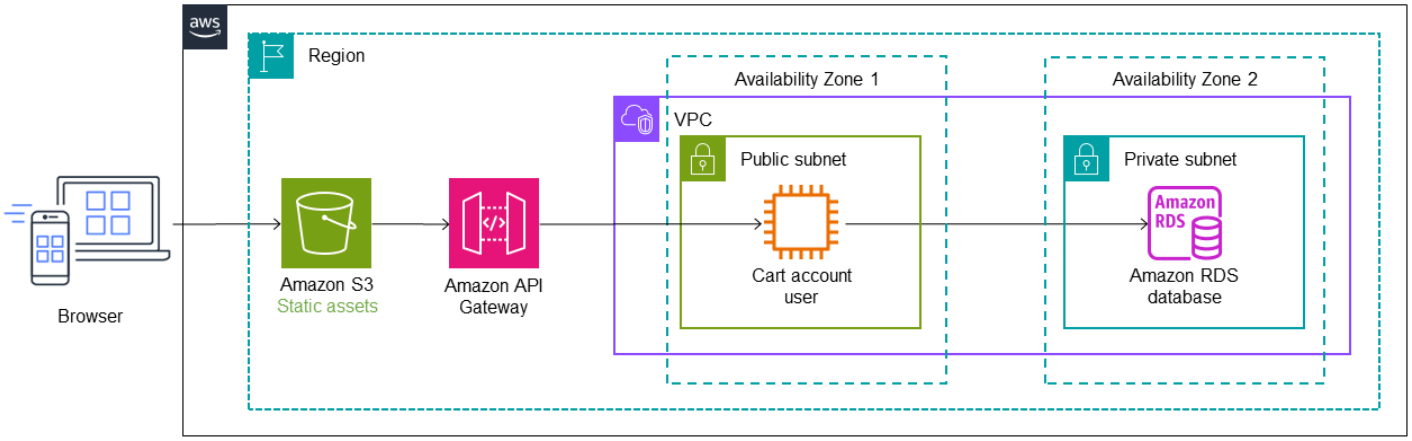
Implementación mediante los servicios de AWS

Uso de API Gateway como proxy de la aplicación

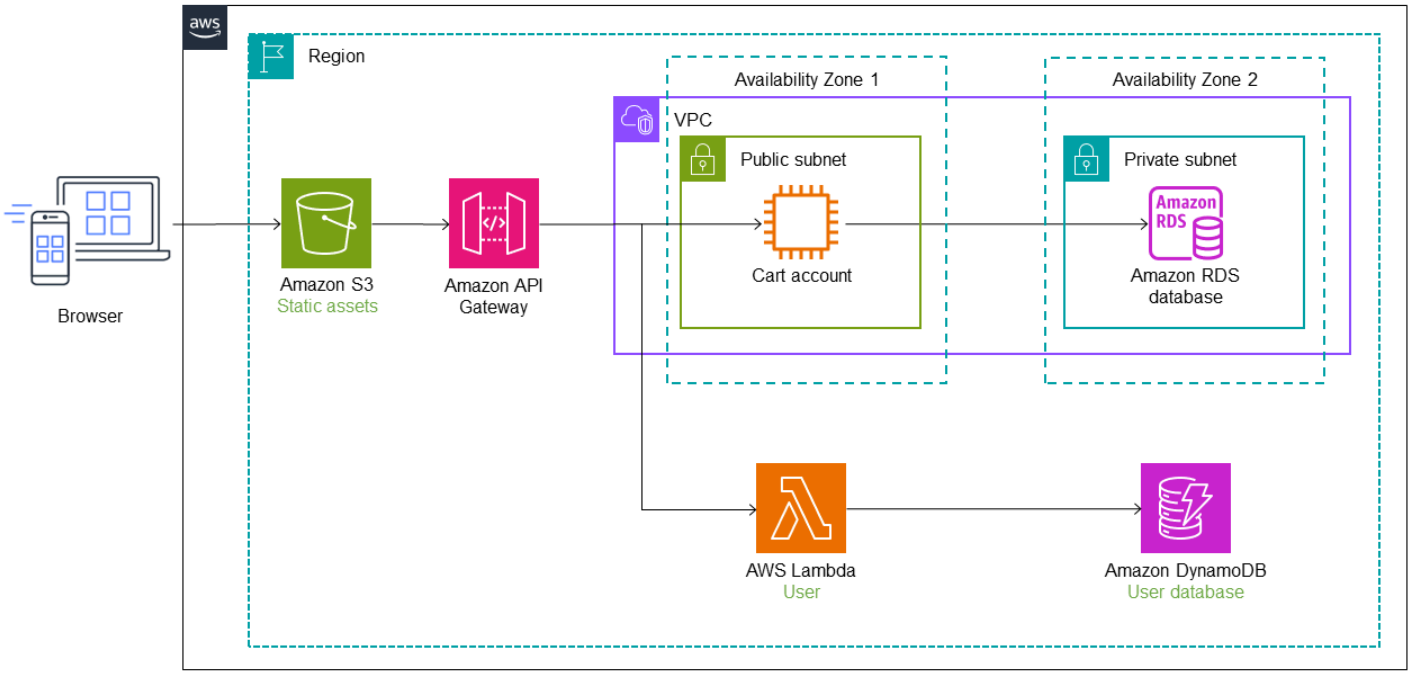
En el diagrama siguiente se ilustra el estado inicial de la aplicación monolítica. Supongamos que se migró a AWS con una estrategia de lift-and-shift, por lo que se ejecuta en una instancia de Amazon Elastic Compute Cloud (Amazon EC2) y usa una base de datos de Amazon Relational Database Service (Amazon RDS). Para simplificar, la arquitectura utiliza una nube privada virtual (VPC) única con una subred privada y otra pública, y supongamos que los microservicios se implementarán desde un inicio en la misma Cuenta de AWS. (La práctica recomendada en los entornos de producción es utilizar una arquitectura de varias cuentas para garantizar la independencia de la implementación). La instancia de EC2 reside en una zona de disponibilidad única de la subred pública y la instancia de RDS reside en una zona de disponibilidad única de la subred privada. Amazon Simple Storage Service (Amazon S3) almacena activos estáticos como los archivos de JavaScript, CSS y React para el sitio web.



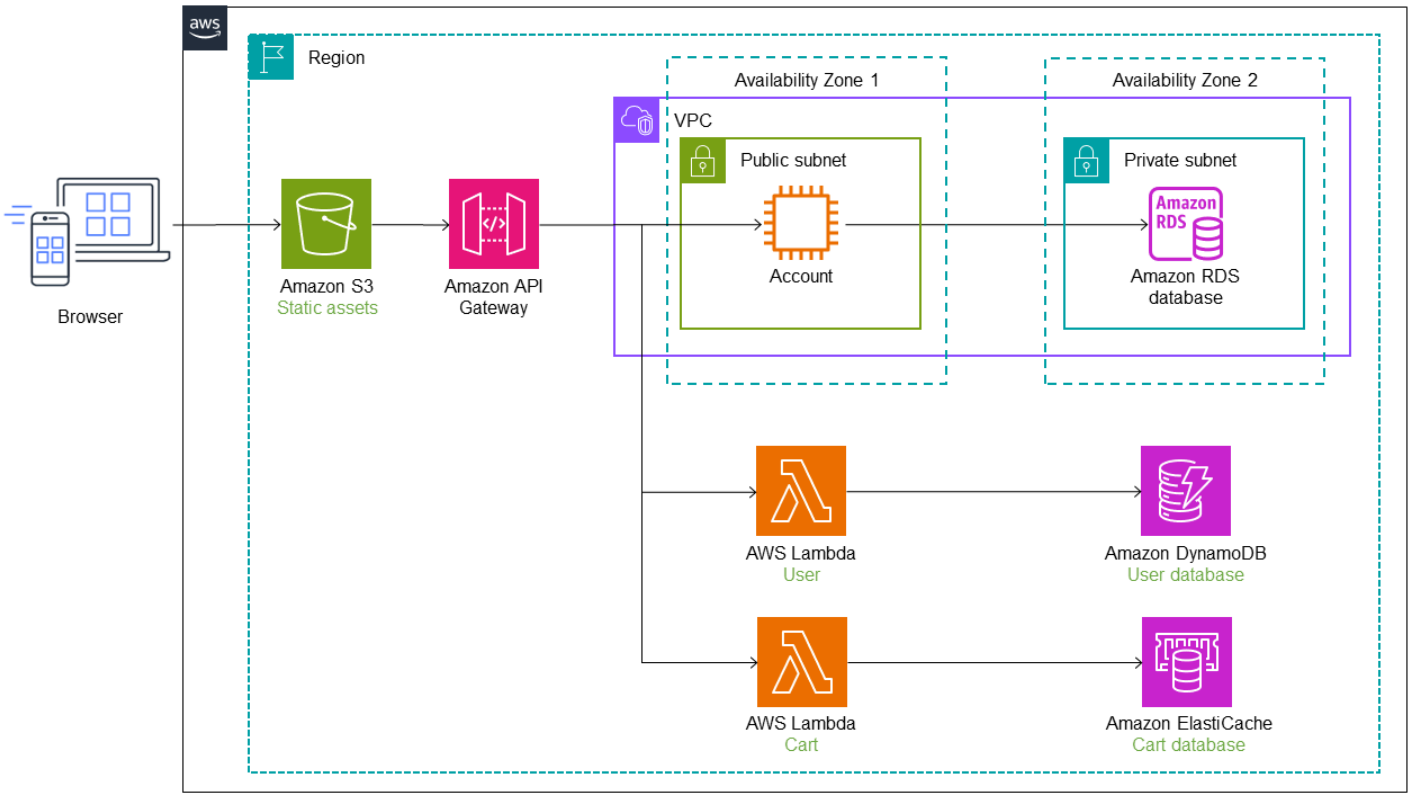
En la arquitectura siguiente, AWS Migration Hub Refactor Spaces implementa Amazon API Gateway delante de la aplicación monolítica. Refactor Spaces crea una infraestructura de refactorización dentro de la cuenta y API Gateway actúa como capa del proxy para enrutar las llamadas al monolito. Desde un inicio, todas las llamadas se enrutan a la aplicación monolítica a través de la capa del proxy. Como se mencionó anteriormente, las capas del proxy pueden convertirse en un punto de error único. Sin embargo, el uso de API Gateway como proxy mitiga el riesgo, ya que se trata de un servicio multi-AZ sin servidor.



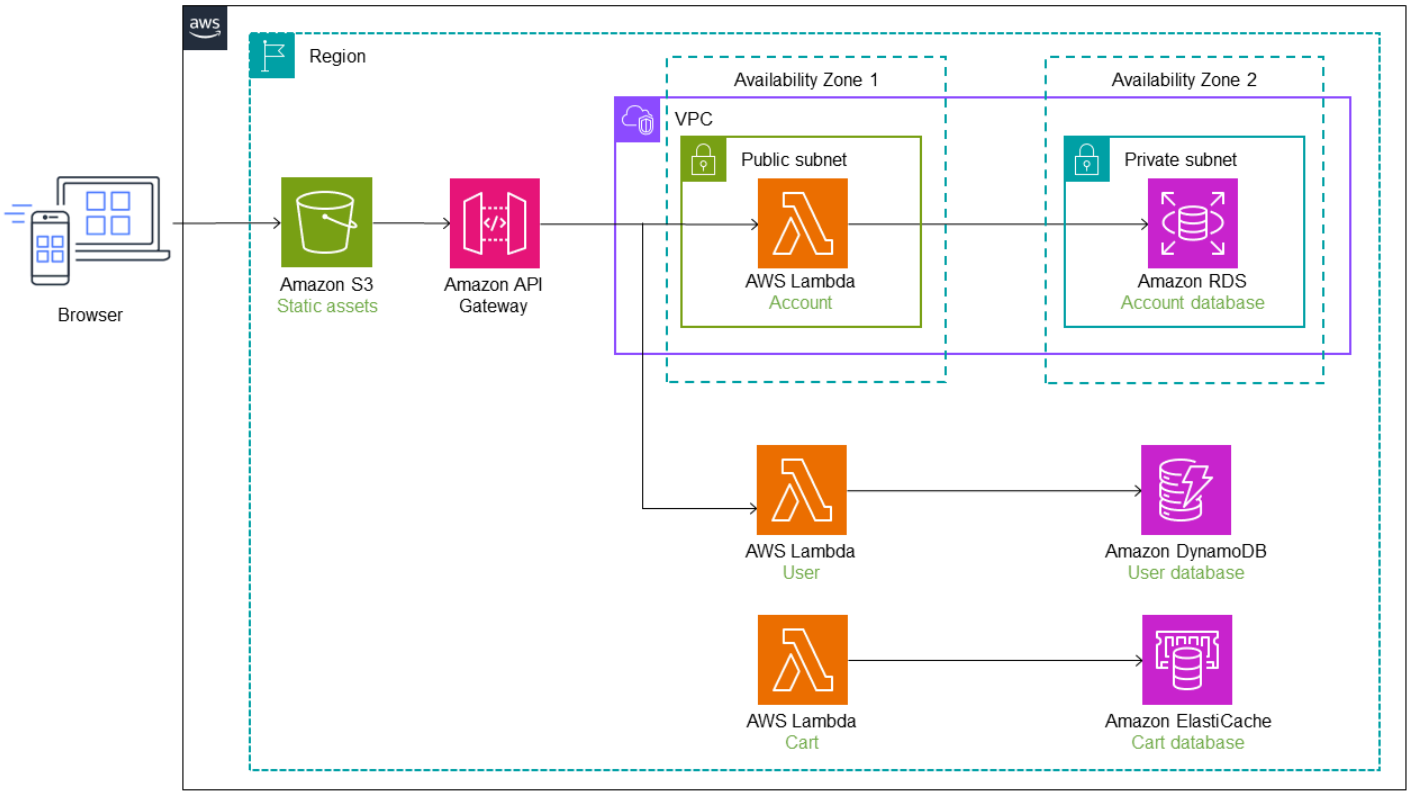
El servicio de usuario se migra a una función de Lambda y una base de datos de Amazon DynamoDB almacena sus datos. Se agregan un punto de conexión del servicio de Lambda y una ruta predeterminada a Refactor Spaces. API Gateway se configura de manera automática para enrutar las llamadas a la función de Lambda.



En el diagrama siguiente, el servicio de carrito también se migró del monolito a una función de Lambda. Se agregan una ruta y un punto de conexión de servicio adicionales a Refactor Spaces. El tráfico pasa de manera automática a la función de Lambda `Cart`. Amazon ElastiCache administra el almacén de datos de la función de Lambda. La aplicación monolítica aún permanece en la instancia de EC2 junto con la base de datos de Amazon RDS.



En el diagrama siguiente, el último servicio (cuenta) se migra del monolito a una función de Lambda. Sigue utilizando la base de datos de Amazon RDS original. Ahora, la arquitectura nueva tiene tres microservicios con bases de datos independientes. Cada servicio utiliza un tipo de base de datos distinto. Este concepto de uso de bases de datos personalizadas para satisfacer las necesidades específicas de los microservicios se denomina persistencia políglota. Las funciones de Lambda también se pueden implementar en distintos lenguajes de programación, según lo determine el caso de uso. Durante la refactorización, Refactor Spaces automatiza la transición y el enrutamiento del tráfico a Lambda. Esto ahorra a los desarrolladores el tiempo necesario para diseñar, implementar y configurar la infraestructura de enrutamiento.



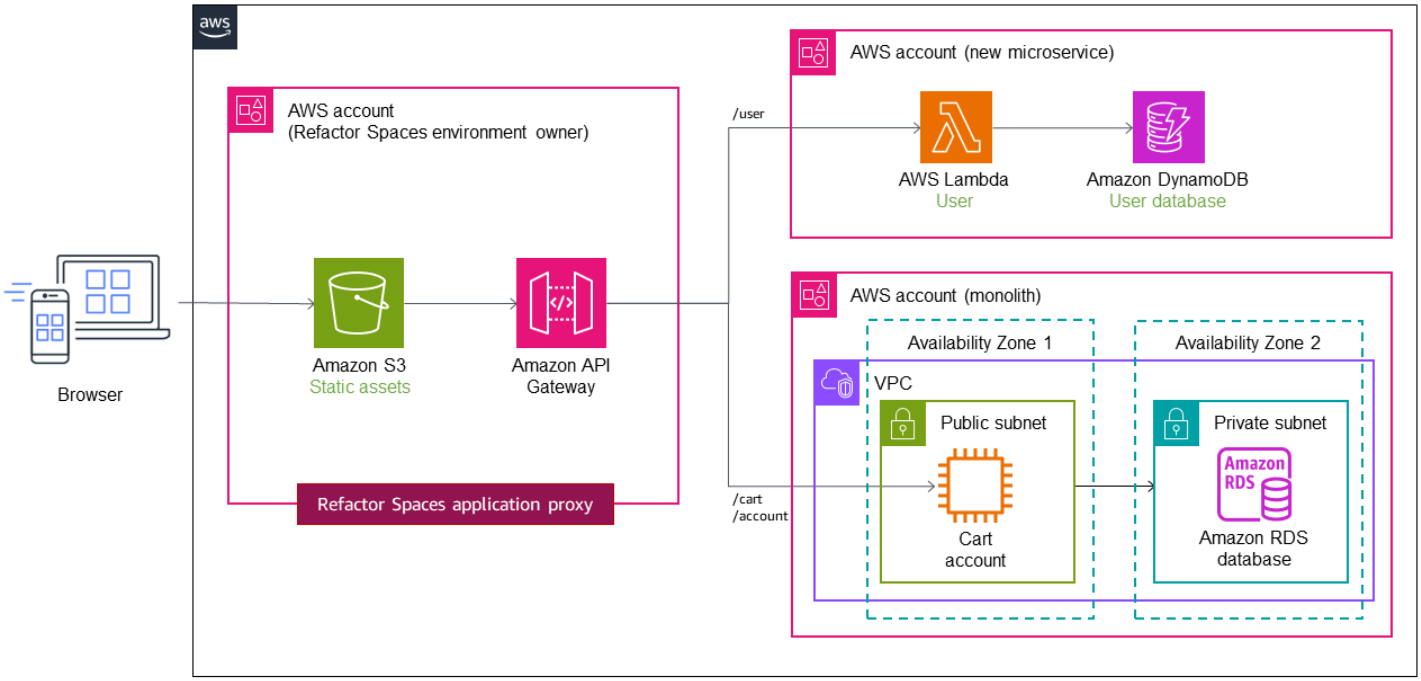
Uso de varias cuentas

En la implementación anterior, utilizamos una VPC única con una subred privada y una pública para la aplicación monolítica, e implementamos los microservicios en la misma Cuenta de AWS por motivos de simplicidad. Sin embargo, esto es poco frecuente en situaciones reales, en las que los microservicios se suelen implementar en varias Cuentas de AWS para lograr una implementación independiente. En una estructura de varias cuentas, es necesario configurar el tráfico de enrutamiento desde el monolito a los servicios nuevos en cuentas distintas.

Refactor Spaces ayuda a crear y configurar la infraestructura de AWS para que las llamadas de la API se desvíen de la aplicación monolítica. Refactor Spaces orquesta políticas de API Gateway, Equilibrador de carga de red y basadas en recursos de AWS Identity and Access Management (IAM) dentro de sus cuentas de AWS como parte de su recurso de aplicación. Puede agregar servicios nuevos de manera transparente en una Cuenta de AWS única o en varias cuentas a un punto de conexión HTTP externo. Todos estos recursos están organizados dentro de Cuenta de AWS y se pueden personalizar y configurar después de la implementación.

Supongamos que los servicios de usuario y de carrito se implementan en dos cuentas distintas, como se muestra en el diagrama siguiente. Cuando utiliza Refactor Spaces, solo es necesario configurar el punto de conexión del servicio y la ruta. Refactor Spaces automatiza la integración entre

API Gateway y Lambda y la creación de políticas de recursos de Lambda, para que pueda centrarse en refactorizar los servicios de manera segura fuera del monolito.



Para ver un tutorial en video sobre el uso de Refactor Spaces, consulte Refactor Apps Incrementally with AWS Migration Hub Refactor Spaces.

Taller

- Iterative app modernization workshop

Referencias de blogs

- AWS Migration Hub Refactor Spaces
- Deep Dive on an AWS Migration Hub Refactor Spaces
- Deployment Pipelines Reference Architecture and Reference Implementations

Contenido relacionado

- Patrones de enrutamiento de API
- Documentación sobre Refactor Spaces

Patrón de bandeja de salida transaccional

Intención

El patrón de bandeja de salida transaccional resuelve el problema de las operaciones de escritura dual que se produce en los sistemas distribuidos cuando una sola operación implica tanto una operación de escritura en la base de datos como una notificación de mensaje o evento. Una operación de escritura dual se produce cuando una aplicación escribe en dos sistemas diferentes; por ejemplo, cuando un microservicio necesita conservar los datos en la base de datos y enviar un mensaje para notificar a otros sistemas. Si se produce un error en una de estas operaciones, es posible que los datos no sean coherentes.

Motivación

Cuando un microservicio envía una notificación de evento tras una actualización de la base de datos, estas dos operaciones deben ejecutarse de forma atómica para garantizar la coherencia y la fiabilidad de los datos.

- Si la actualización de la base de datos se realiza correctamente pero la notificación del evento falla, el servicio descendente no se dará cuenta del cambio y el sistema podría entrar en un estado incoherente.
- Si se produce un error en la actualización de la base de datos, pero se envía la notificación del evento, los datos podrían dañarse, lo que podría afectar a la fiabilidad del sistema.

Aplicabilidad

Utilice el patrón de bandeja de salida transaccional cuando:

- Está creando una aplicación basada en eventos en la que una actualización de la base de datos inicia la notificación de un evento.
- Desea garantizar la atomicidad en las operaciones que implican dos servicios.
- Desea implementar el patrón de aprovisionamiento de eventos.

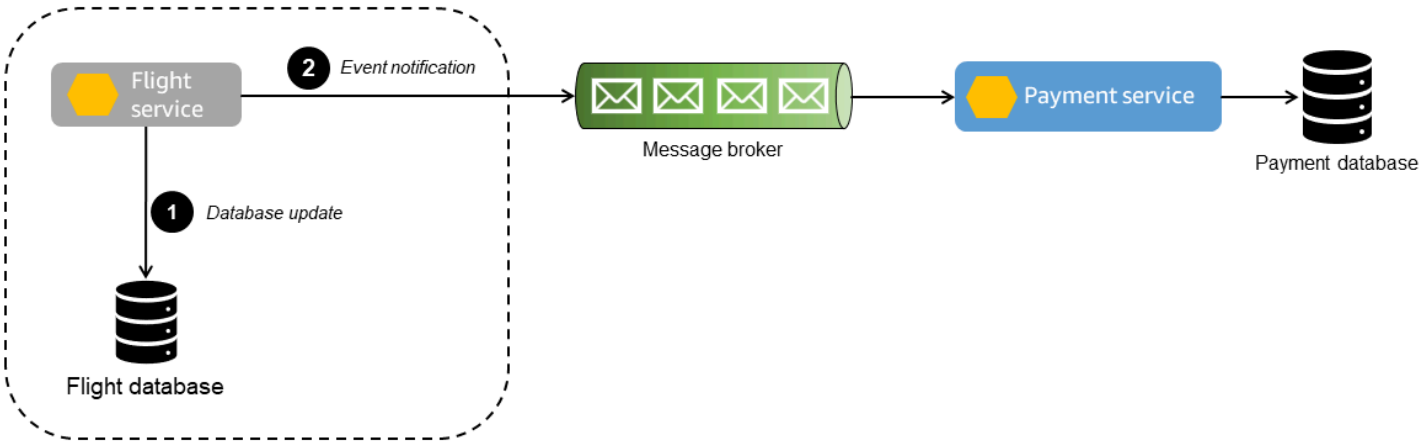
Problemas y consideraciones

- Mensajes duplicados: el servicio de procesamiento de eventos puede enviar mensajes o eventos duplicados, por lo que le recomendamos que haga que el servicio consumidor sea idempotente mediante el seguimiento de los mensajes procesados.
- Orden de notificación: envíe los mensajes o eventos en el mismo orden en que el servicio actualiza la base de datos. Esto es fundamental para el patrón de aprovisionamiento de eventos, ya que se puede utilizar un almacén de eventos para recuperar el almacén de datos en un momento dado. Si el orden es incorrecto, podría poner en peligro la calidad de los datos. La coherencia eventual y la reversión de la base de datos pueden agravar el problema si no se mantiene el orden de las notificaciones.
- Reversión de la transacción: no envíe una notificación de evento si la transacción se ha revertido.
- Gestión de transacciones a nivel de servicio: si la transacción abarca servicios que requieren actualizaciones del almacén de datos, utilice el patrón de orquestación de la saga para preservar la integridad de los datos en todos los almacenes de datos.

Implementación

Arquitectura de alto nivel

El siguiente diagrama de secuencia muestra el orden de los eventos que se producen durante las operaciones de escritura dual.



1. El servicio de vuelo escribe en la base de datos y envía una notificación de evento al servicio de pago.

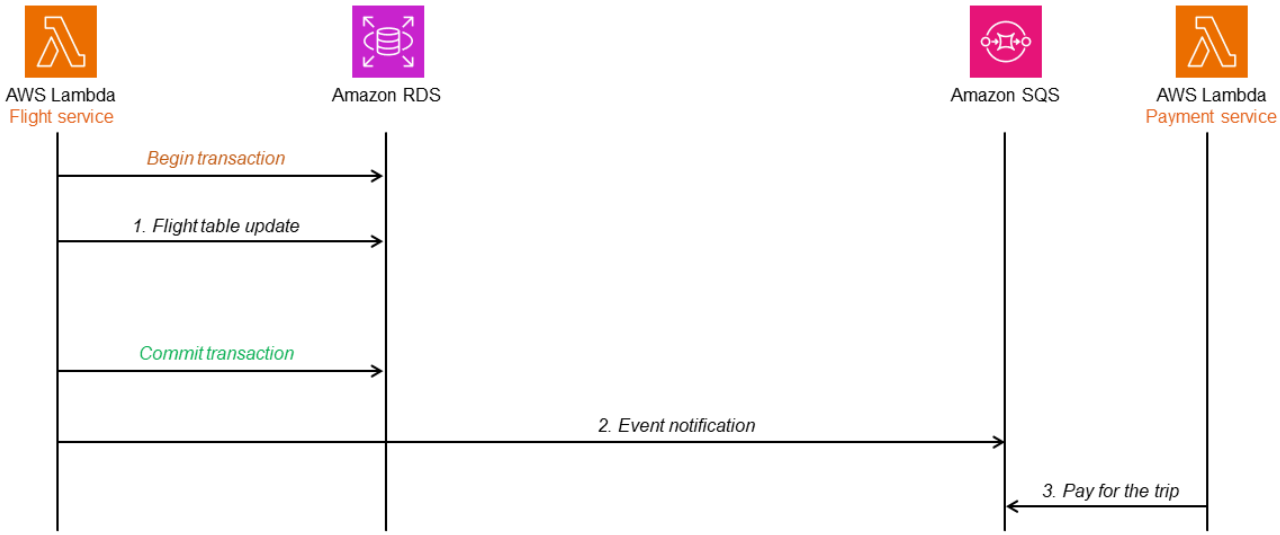
2. El agente de mensajes lleva los mensajes y eventos al servicio de pago. Cualquier fallo en el agente de mensajes impide que el servicio de pago reciba las actualizaciones.

Si se produce un error en la actualización de la base de datos de vuelos, pero se envía la notificación, el servicio de pago procesará el pago en función de la notificación del evento. Esto provocará incoherencias en los datos posteriores.

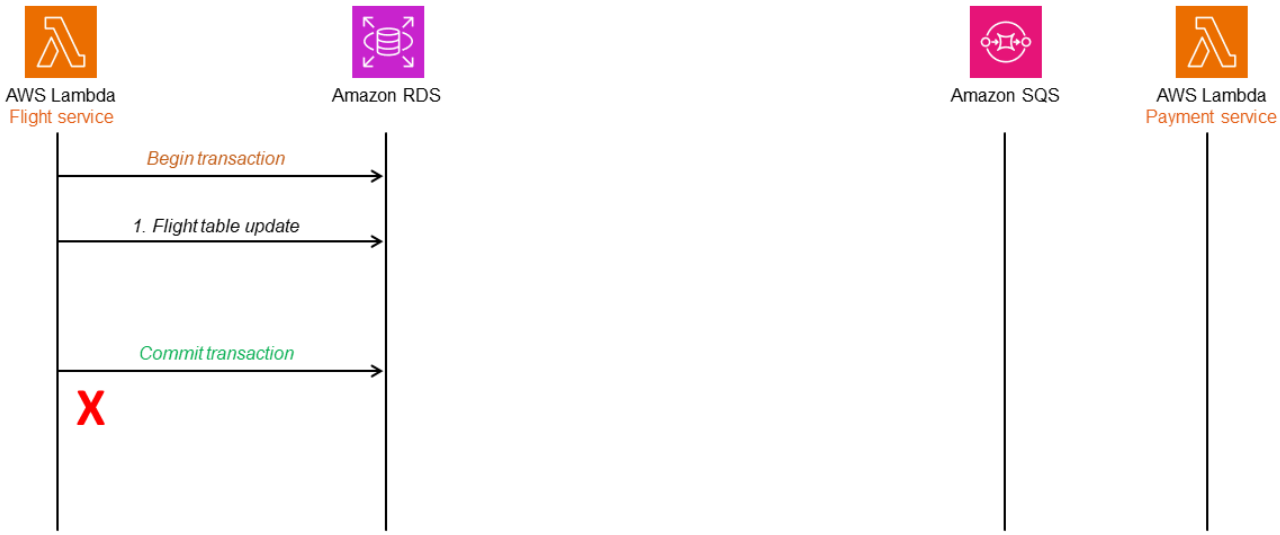
Implementación mediante los servicios de AWS

Para demostrar el patrón en el diagrama de secuencia, utilizaremos los servicios siguientes de AWS, como se muestra en el diagrama siguiente.

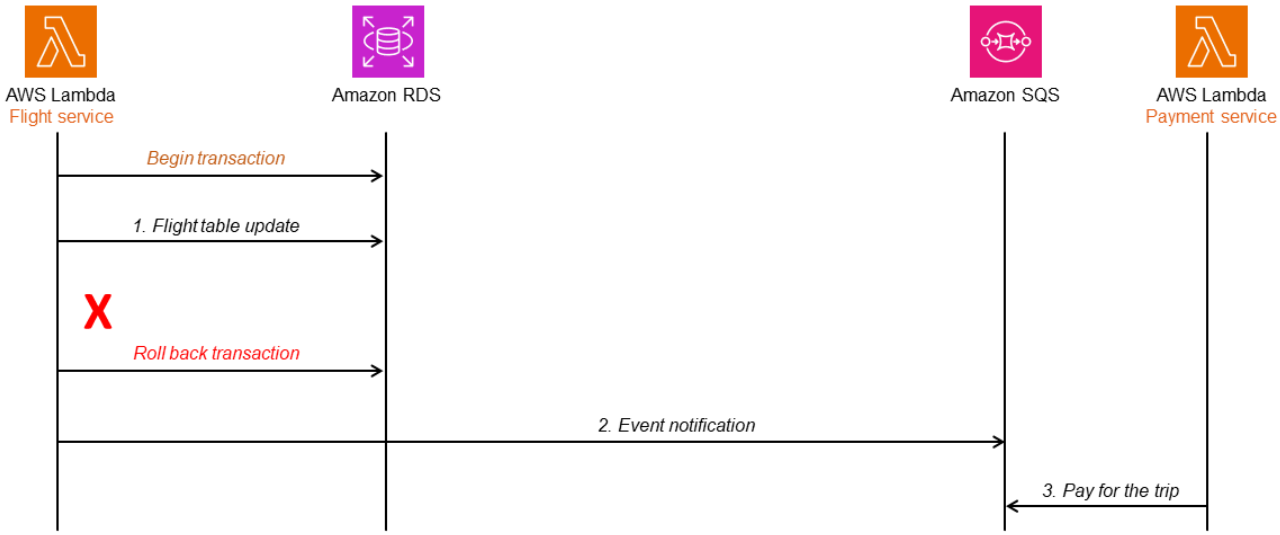
- Los microservicios se implementan mediante el uso de AWS Lambda.
- La base de datos principal está administrada por Amazon Relational Database Service (Amazon RDS).
- Amazon Simple Queue Service (Amazon SQS) actúa como agente de mensajes que recibe las notificaciones de eventos.



Si se produce un error en el servicio de vuelo después de confirmar la transacción, es posible que no se envíe la notificación del evento.



Sin embargo, la transacción podría fallar y revertirse, pero la notificación del evento podría seguir enviándose y provocar que el servicio de pago procese el pago.



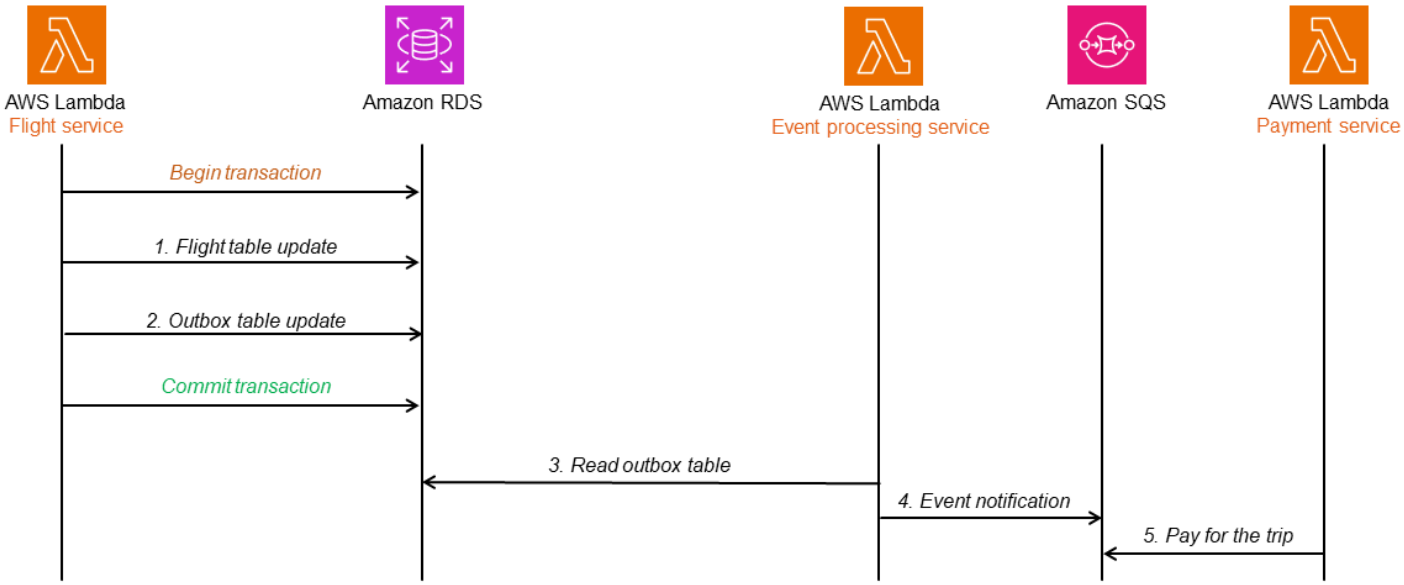
Para solucionar este problema, puede utilizar una tabla de bandeja de salida o cambiar la captura de datos (CDC). En las siguientes secciones se analizan estas dos opciones y cómo puede implementarlas mediante los servicios de AWS.

Uso de una tabla de bandeja de salida con una base de datos relacional

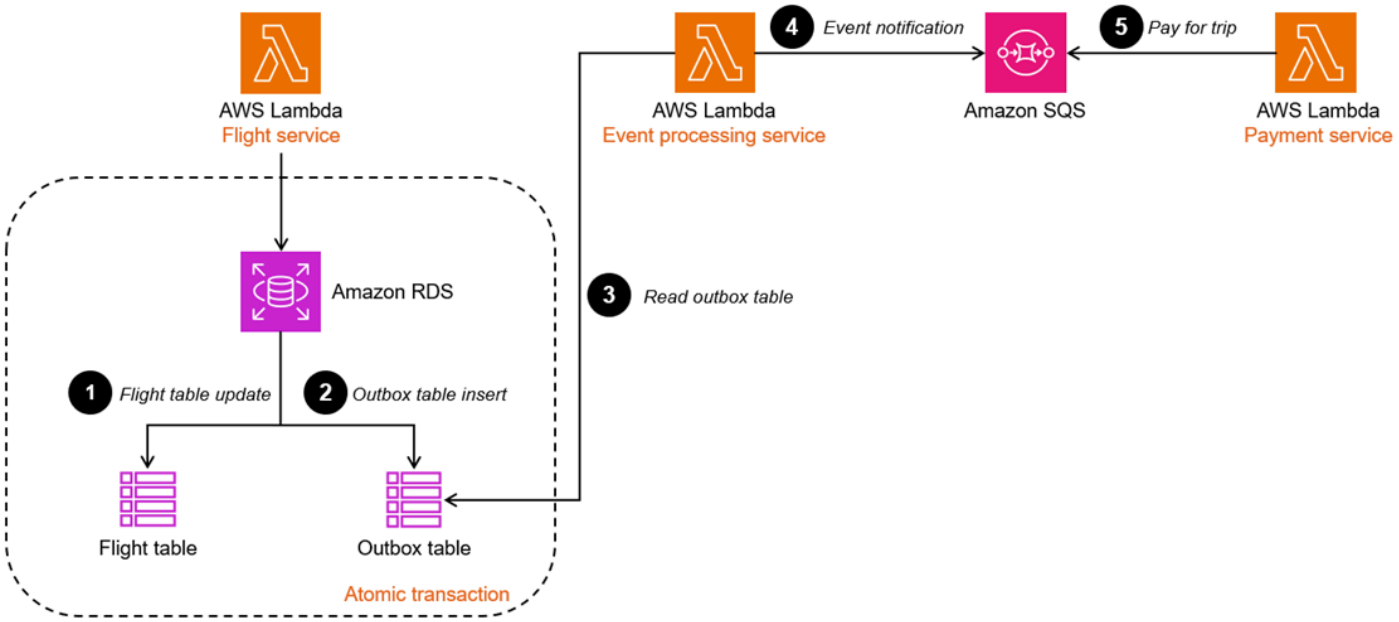
Una tabla de bandeja de salida almacena todos los eventos del servicio de vuelo con una marca de tiempo y un número de secuencia.

Cuando se actualiza la tabla de vuelos, la tabla de bandeja de salida también se actualiza en la misma transacción. Otro servicio (por ejemplo, el servicio de procesamiento de eventos) lee la tabla de la bandeja de salida y envía el evento a Amazon SQS. Amazon SQS envía un mensaje sobre el evento al servicio de pago para su posterior procesamiento. Las colas estándar de Amazon SQS garantizan que el mensaje se entregue al menos una vez y no se pierda. Sin embargo, cuando utiliza las colas estándar de Amazon SQS, es posible que el mismo mensaje o evento se entregue más de una vez, por lo que debe asegurarse de que el servicio de notificación de eventos sea idempotente (es decir, procesar el mismo mensaje varias veces no debería tener un efecto adverso). Si necesita que el mensaje se entregue exactamente una vez, al ordenar los mensajes, puede utilizar las colas FIFO (primero en entrar, primero en salir) de Amazon SQS.

Si se produce un error en la actualización de la tabla de vuelos o en la actualización de la tabla de bandeja de salida, se revierte toda la transacción para que no haya inconsistencias en los datos posteriores.



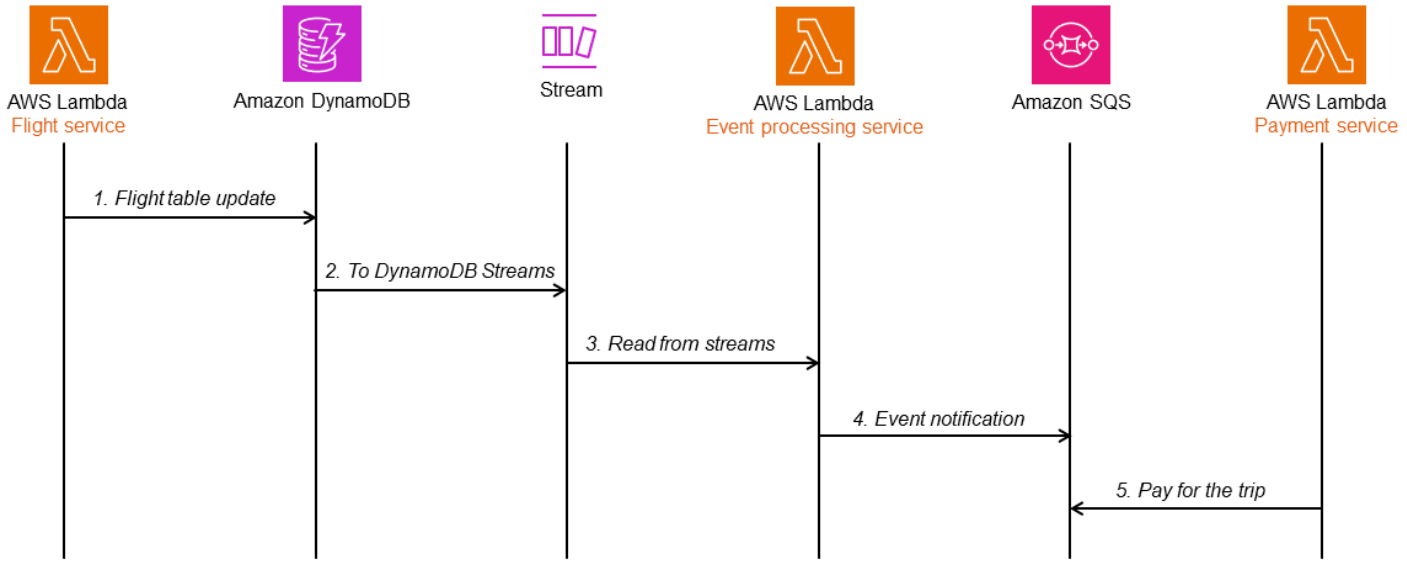
En el siguiente diagrama, la arquitectura de la bandeja de salida transaccional se implementa mediante una base de datos de Amazon RDS. Cuando el servicio de procesamiento de eventos lee la tabla de bandeja de salida, solo reconoce las filas que forman parte de una transacción confirmada (correcta) y, a continuación, coloca el mensaje del evento en la cola de SQS, que el servicio de pago lee para su posterior procesamiento. Este diseño resuelve el problema de las operaciones de escritura dual y preserva el orden de los mensajes y eventos mediante el uso de marcas de tiempo y números de secuencia.



Uso de la captura de datos de cambio (CDC)

Algunas bases de datos admiten la publicación de modificaciones a nivel de elemento para capturar los datos modificados. Puede identificar los elementos modificados y enviar una notificación de evento en consecuencia. Esto ahorra la sobrecarga de crear otra tabla para realizar un seguimiento de las actualizaciones. El evento iniciado por el servicio de vuelo se almacena en otro atributo del mismo artículo.

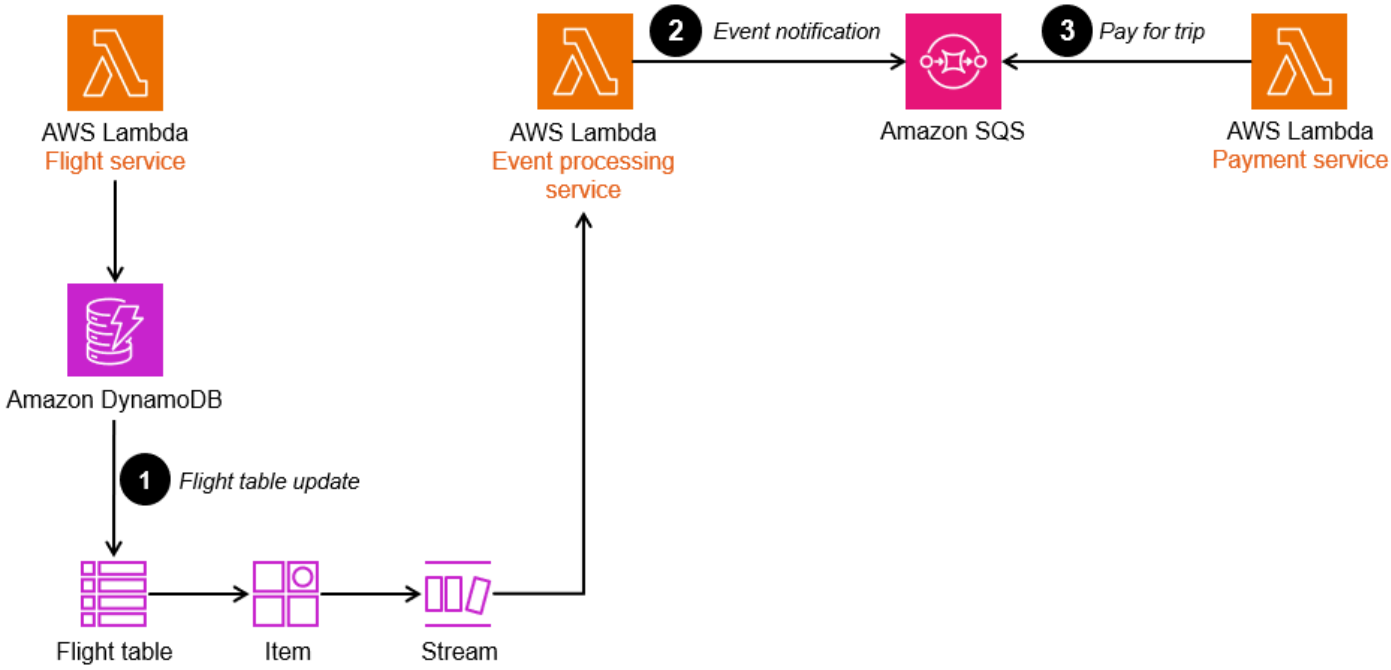
Amazon DynamoDB es una base de datos NoSQL de clave-valor que admite actualizaciones de CDC. En el siguiente diagrama secuencial, DynamoDB publica las modificaciones a nivel de elemento en Amazon DynamoDB Streams. El servicio de procesamiento de eventos lee las transmisiones y publica la notificación del evento en el servicio de pago para su posterior procesamiento.



DynamoDB Streams captura el flujo de información relacionado con los cambios a nivel de elemento en una tabla de DynamoDB mediante una secuencia ordenada por tiempo.

Para implementar un patrón de bandeja de salida transaccional, habilite las transmisiones en la tabla de DynamoDB. La función de Lambda del servicio de procesamiento de eventos está asociada a estos flujos.

- Cuando se actualiza la tabla de vuelos, DynamoDB Streams captura los datos modificados y el servicio de procesamiento de eventos sondea la transmisión en busca de nuevos registros.
- Cuando hay nuevos registros de transmisión disponibles, la función de Lambda coloca de forma sincrónica el mensaje del evento en la cola de SQS para su posterior procesamiento. Puede agregar un atributo al elemento de DynamoDB para capturar la marca de tiempo y el número de secuencia según sea necesario para mejorar la solidez de la implementación.



Código de muestra

Uso de una tabla de bandeja de salida

En el código de ejemplo de esta sección se muestra cómo puede implementar el patrón de bandeja de salida transaccional mediante una tabla de bandeja de salida. Para ver el código completo, consulte el repositorio de GitHub para ver este ejemplo.

El siguiente fragmento de código guarda la entidad `Flight` y el evento `Flight` de la base de datos en sus tablas respectivas dentro de una sola transacción.

```
@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;
    }
```

Un servicio independiente se encarga de escanear periódicamente la tabla de bandeja de salida en busca de nuevos eventos, enviarlos a Amazon SQS y eliminarlos de la tabla si Amazon SQS responde correctamente. La tasa de sondeo se puede configurar en el archivo `application.properties`.

```
@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);
        }
    }
```

Uso de la captura de datos de cambio (CDC)

En el código de ejemplo de esta sección se muestra cómo puede implementar el patrón de bandeja de salida transaccional mediante las funcionalidades del registro de datos de cambios (CDC) de DynamoDB. Para ver el código completo, consulte el repositorio de GitHub para ver este ejemplo.

En el fragmento siguiente de código de AWS Cloud Development Kit (AWS CDK) se crea una tabla de vuelos de DynamoDB y un flujo de datos de Amazon Kinesis (`cdcStream`), y configura la tabla de vuelos para enviar todas sus actualizaciones al flujo.

```
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,
    }

});
```

En el fragmento de código y la configuración siguientes se define una función de flujo de Spring Cloud que toma las actualizaciones de la transmisión de Kinesis y reenvía estos eventos a una cola de SQS para su procesamiento posterior.

```
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);
    }
}
```

Repositorio GitHub

Para obtener una implementación completa de la arquitectura de ejemplo para este patrón, consulte el repositorio de GitHub en https://github.com/aws-samples/transactional-outbox-pattern.

Recursos

Referencias

- AWS Centro de arquitectura de
- AWS Centro de desarrolladores de
- The Amazon Builders Library

Herramientas

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

Metodologías

- The Twelve-Factor App (ePub by Adam Wiggins)
- Nygard, Michael T. Release It!: Design and Deploy Production-Ready Software. 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2018
- Polyglot Persistence (entrada de blog de Martin Fowler)
- StranglerFigApplication (entrada de blog de Martin Fowler)

Historial de documentos

En la siguiente tabla, se describen cambios significativos de esta guía. Si quiere recibir notificaciones de futuras actualizaciones, puede suscribirse a las notificaciones RSS.

Cambio	Descripción	Fecha
Nuevos patrones	Se agregaron dos patrones nuevos: arquitectura hexagonal y dispersión y recopilación.	7 de mayo de 2024
Nuevos ejemplos de código	Se agregó un código de muestra para el caso de uso de la captura de datos de cambios (CDC) al patrón de la bandeja de salida transacci onal.	23 de febrero de 2024
Nuevos ejemplos de código	• Se ha actualizado el patrón de bandeja de salida transaccional con un código de muestra. • Se ha eliminado la sección dedicada a los patrones de orquestación y coreografía, que ha sido sustituida por la coreografía de la saga y la orquestación de la saga.	16 de noviembre de 2023
Nuevos patrones	Se han agregado tres patrones nuevos: coreografía de la saga, publicación/suscri pción y aprovisionamiento de eventos.	14 de noviembre de 2023

Actualización de	Se ha actualizado la sección Implementación del patrón de higo estrangulador.	2 de octubre de 2023
Publicación inicial	Esta primera versión incluye ocho patrones de diseño: capa anticorrupción (ACL), enrutamiento de API, interruptor, orquestación y coreografía, reintento con retroceso, orquestación de la saga, higo estrangulador y bandeja de salida transaccional.	28 de julio de 2023

Glosario de las Recomendaciones de AWS

Los siguientes son términos de uso común en las estrategias, guías y patrones que se ofrecen en las Recomendaciones de AWS. Para sugerir entradas, utilice el enlace Enviar comentarios al final del glosario.

Números

Las 7 R

Siete estrategias de migración comunes para trasladar aplicaciones a la nube. Estas estrategias se basan en las 5 R que Gartner identificó en 2011 y consisten en lo siguiente:

- Refactorizar/rediseñar: traslade una aplicación y modifique su arquitectura mediante el máximo aprovechamiento de las características nativas en la nube para mejorar la agilidad, el rendimiento y la escalabilidad. Por lo general, esto implica trasladar el sistema operativo y la base de datos. Ejemplo: migre la base de datos de Oracle en las instalaciones a la Edición de Amazon Aurora compatible con PostgreSQL.
- Redefinir la plataforma (transportar y redefinir): traslade una aplicación a la nube e introduzca algún nivel de optimización para aprovechar las capacidades de la nube. Ejemplo: migre la base de datos de Oracle en las instalaciones a Amazon Relational Database Service (Amazon RDS) para Oracle en la Nube de AWS.
- Recomprar (readquirir): cambie a un producto diferente, lo cual se suele llevar a cabo al pasar de una licencia tradicional a un modelo SaaS. Ejemplo: migre el sistema de administración de las relaciones con los clientes (CRM) a Salesforce.com.
- Volver a alojar (migrar mediante lift-and-shift): traslade una aplicación a la nube sin realizar cambios para aprovechar las capacidades de la nube. Ejemplo: migre la base de datos de Oracle en las instalaciones a Oracle en una instancia de EC2 en la Nube de AWS.
- Reubicar (migrar el hipervisor mediante lift and shift): traslade la infraestructura a la nube sin comprar equipo nuevo, reescribir aplicaciones o modificar las operaciones actuales. Los servidores se migran de una plataforma en las instalaciones a un servicio en la nube para la misma plataforma. Ejemplo: migre una aplicación de Microsoft Hyper-V a AWS.
- Retener (revisitar): conserve las aplicaciones en el entorno de origen. Estas pueden incluir las aplicaciones que requieren una refactorización importante, que desee posponer para más adelante, y las aplicaciones heredadas que desee retener, ya que no hay ninguna justificación empresarial para migrarlas.

- Retirar: retire o elimine las aplicaciones que ya no sean necesarias en un entorno de origen.

A

ABAC

Consulte control de acceso basado en atributos.

servicios abstractos

Consulte servicios administrados.

ACID

Consulte atomicidad, consistencia, aislamiento, durabilidad.

migración activa-activa

Método de migración de bases de datos en el que las bases de datos de origen y destino se mantienen sincronizadas (mediante una herramienta de replicación bidireccional o mediante operaciones de escritura doble) y ambas bases de datos gestionan las transacciones de las aplicaciones conectadas durante la migración. Este método permite la migración en lotes pequeños y controlados, en lugar de requerir una transición única. Es más flexible, pero requiere más trabajo que una migración activa-pasiva.

migración activa-pasiva

Método de migración de bases de datos en el que las bases de datos de origen y destino se mantienen sincronizadas, pero solo la de origen gestiona las transacciones de las aplicaciones conectadas, mientras los datos se replican en la de destino. La base de datos de destino no acepta ninguna transacción durante la migración.

función de agregación

Función SQL que actúa en un grupo de filas y calcula un único valor de devolución para el grupo. Entre los ejemplos de funciones de agregación se incluyen `SUM` y `MAX`.

IA

Consulte inteligencia artificial.

AIOps

Consulte operaciones de inteligencia artificial

anonimización

El proceso de eliminar permanentemente la información personal de un conjunto de datos. La anonimización puede ayudar a proteger la privacidad personal. Los datos anonimizados ya no se consideran datos personales.

antipatrones

Una solución que se utiliza con frecuencia para un problema recurrente en el que la solución es contraproducente, ineficaz o menos eficaz que una alternativa.

control de aplicaciones

Enfoque de seguridad que permite usar exclusivamente aplicaciones aprobadas para ayudar a proteger un sistema contra el malware.

cartera de aplicaciones

Recopilación de información detallada sobre cada aplicación que utiliza una organización, incluido el costo de creación y mantenimiento de la aplicación y su valor empresarial. Esta información es clave para el proceso de detección y análisis de la cartera y ayuda a identificar y priorizar las aplicaciones que se van a migrar, modernizar y optimizar.

inteligencia artificial (IA)

El campo de la informática que se dedica al uso de tecnologías informáticas para realizar funciones cognitivas que suelen estar asociadas a los seres humanos, como el aprendizaje, la resolución de problemas y el reconocimiento de patrones. Para más información, consulte ¿Qué es la inteligencia artificial?

operaciones de inteligencia artificial (AIOps)

El proceso de utilizar técnicas de machine learning para resolver problemas operativos, reducir los incidentes operativos y la intervención humana, y mejorar la calidad del servicio. Para obtener más información sobre cómo se utiliza AIOps en la estrategia de migración de AWS, consulte la Guía de integración de operaciones.

cifrado asimétrico

Algoritmo de cifrado que utiliza un par de claves, una clave pública para el cifrado y una clave privada para el descifrado. Puede compartir la clave pública porque no se utiliza para el descifrado, pero el acceso a la clave privada debe estar sumamente restringido.

atomicidad, consistencia, aislamiento, durabilidad (ACID)

Conjunto de propiedades de software que garantizan la validez de los datos y la fiabilidad operativa de una base de datos, incluso en caso de errores, cortes de energía u otros problemas.

control de acceso basado en atributos (ABAC)

La práctica de crear permisos detallados basados en los atributos del usuario, como el departamento, el puesto de trabajo y el nombre del equipo. A fin de obtener más información, consulte ABAC para AWS en la documentación de AWS Identity and Access Management (IAM).

origen de datos fidedigno

Ubicación en la que se almacena la versión principal de los datos, que se considera la fuente de información más fiable. Puede copiar los datos del origen de datos autorizado a otras ubicaciones con el fin de procesarlos o modificarlos, por ejemplo, anonimizarlos, redactarlos o seudonimizarlos.

Zona de disponibilidad

Ubicación diferenciada de una Región de AWS que está aislada de los errores que se producen en otras zonas de disponibilidad y que brinda conectividad de red económica y de baja latencia a otras zonas de disponibilidad de la misma región.

AWS Cloud Adoption Framework (AWS CAF)

Marco de directrices y prácticas recomendadas de AWS para ayudar a las empresas a desarrollar un plan eficiente y eficaz a fin de migrar con éxito a la nube de AWS. CAF organiza la orientación en seis áreas de enfoque llamadas perspectivas: empresarial, humana, gobernanza, plataforma, seguridad y operaciones. Las perspectivas empresariales, humanas y de gobernanza se centran en las habilidades y los procesos empresariales; las perspectivas de plataforma, seguridad y operaciones se centran en las habilidades y los procesos técnicos. Por ejemplo, la perspectiva humana se dirige a las partes interesadas que se ocupan de los Recursos Humanos (RR. HH.), las funciones del personal y la administración de las personas. Desde esta perspectiva, AWS CAF brinda orientación para el desarrollo, la capacitación y la comunicación de las personas, con el fin de ayudar a preparar la organización para una adopción exitosa de la nube. Para obtener más información, consulte la Página web de AWS CAF y el Documento técnico de AWS CAF.

AWS Workload Qualification Framework (AWS WQF)

Herramienta que evalúa las cargas de trabajo de migración de bases de datos, recomienda estrategias de migración y brinda estimaciones de trabajo. AWS WQF se incluye con AWS

Schema Conversion Tool (AWS SCT). Analiza los esquemas de bases de datos y los objetos de código, el código de las aplicaciones, las dependencias y las características de rendimiento y proporciona informes de evaluación.

B

bot malicioso

Bot destinado a causar interrupciones o daños a personas u organizaciones.

BCP

Consulte planificación de la continuidad del negocio.

gráfico de comportamiento

Una vista unificada e interactiva del comportamiento de los recursos y de las interacciones a lo largo del tiempo. Puede utilizar un gráfico de comportamiento con Amazon Detective para examinar los intentos de inicio de sesión fallidos, las llamadas sospechosas a la API y acciones similares. Para obtener más información, consulte Datos en un gráfico de comportamiento en la documentación de Detective.

sistema big-endian

Un sistema que almacena primero el byte más significativo. Consulte también endianidad.

clasificación binaria

Un proceso que predice un resultado binario (una de las dos clases posibles). Por ejemplo, es posible que su modelo de ML necesite predecir problemas como “¿Este correo electrónico es spam o no es spam?” o “¿Este producto es un libro o un automóvil?”.

filtro de floración

Estructura de datos probabilística y eficiente en términos de memoria que se utiliza para comprobar si un elemento es miembro de un conjunto.

implementación azul/verde

Estrategia de implementación en la que se crean dos entornos separados, pero idénticos. La versión actual de la aplicación se ejecuta en un entorno (azul) y la nueva versión de la aplicación se ejecuta en el otro entorno (verde). Esta estrategia lo ayuda a hacer reversiones rápidamente con un impacto mínimo.

bot

Aplicación de software que ejecuta tareas automatizadas a través de Internet y simula la actividad o interacción humana. Algunos bots son útiles o beneficiosos, como los rastreadores web que indexan la información de internet. Otros bots, conocidos como bots maliciosos, tienen como objetivo causar interrupciones o daños a personas u organizaciones.

botnet

Redes de bots infectadas por malware y que están bajo el control de una sola parte, conocida como pastor de bots u operador de bots. Las botnets son el mecanismo más conocido para escalar los bots y su impacto.

branch

Área contenida de un repositorio de código. La primera rama que se crea en un repositorio es la rama principal. Puede crear una rama nueva a partir de una rama existente y, a continuación, desarrollar características o corregir errores en la rama nueva. Una rama que se genera para crear una característica se denomina comúnmente rama de característica. Cuando la característica se encuentra lista para su lanzamiento, se vuelve a combinar la rama de característica con la rama principal. Para obtener más información, consulte Acerca de las ramas (documentación de GitHub).

acceso de emergencia

En circunstancias excepcionales y mediante un proceso aprobado, un usuario puede acceder rápidamente a una Cuenta de AWS a la que normalmente no tiene permisos para acceder. Para más información, consulte el indicador Implement break-glass procedures en la guía de AWS Well-Architected.

estrategia de implementación sobre infraestructura existente

La infraestructura existente en su entorno. Al adoptar una estrategia de implementación sobre infraestructura existente para una arquitectura de sistemas, se diseña la arquitectura en función de las limitaciones de los sistemas y la infraestructura actuales. Si está ampliando la infraestructura existente, puede combinar las estrategias de implementación sobre infraestructuras existentes y de implementación desde cero.

caché de búfer

El área de memoria donde se almacenan los datos a los que se accede con más frecuencia.

capacidad empresarial

Lo que hace una empresa para generar valor (por ejemplo, ventas, servicio al cliente o marketing). Las arquitecturas de microservicios y las decisiones de desarrollo pueden estar impulsadas por las capacidades empresariales. Para obtener más información, consulte la sección Organizado en torno a las capacidades empresariales del documento técnico Ejecutar microservicios en contenedores en AWS.

planificación de la continuidad del negocio (BCP)

Plan que aborda el posible impacto de un evento disruptivo, como una migración a gran escala en las operaciones y permite a la empresa reanudar las operaciones rápidamente.

C

CAF

Consulte AWS Cloud Adoption Framework.

implementación canario

Lanzamiento lento e incremental de una versión para los usuarios finales. Cuando tenga mayor confianza en la nueva versión, la implementa y reemplaza la versión actual en su totalidad.

CCoE

Consulte Centro de excelencia en la nube.

CDC

Consulte captura de datos de cambios.

captura de datos de cambio (CDC)

Proceso de seguimiento de los cambios en un origen de datos, como una tabla de base de datos, y registro de los metadatos relacionados con el cambio. Puede utilizar los CDC para diversos fines, como auditar o replicar los cambios en un sistema de destino para mantener la sincronización.

ingeniería del caos

Introducción intencionada de fallos o eventos disruptivos para poner a prueba la resiliencia de un sistema. Puede usar AWS Fault Injection Service (AWS FIS) para llevar a cabo experimentos que pongan bajo estrés sus cargas de trabajo de AWS y evalúen su respuesta.

CI/CD

Consulte integración continua y entrega continua.

classification

Un proceso de categorización que permite generar predicciones. Los modelos de ML para problemas de clasificación predicen un valor discreto. Los valores discretos siempre son distintos entre sí. Por ejemplo, es posible que un modelo necesite evaluar si hay o no un automóvil en una imagen.

cifrado del cliente

Cifrado de datos de forma local, antes de que el Servicio de AWS de destino los reciba.

Centro de excelencia en la nube (CCoE)

Equipo multidisciplinario que impulsa los esfuerzos de adopción de la nube en toda la organización, incluido el desarrollo de las prácticas recomendadas en la nube, la movilización de recursos, el establecimiento de plazos de migración y la dirección de la organización durante las transformaciones a gran escala. Para más información, consulte las publicaciones sobre el CCoE en el blog de estrategia empresarial de la Nube de AWS.

computación en la nube

La tecnología en la nube que se utiliza normalmente para la administración de dispositivos de IoT y el almacenamiento de datos de forma remota. La computación en la nube suele estar relacionada con la tecnología de computación de periferia.

modelo operativo en la nube

En una organización de TI, el modelo operativo que se utiliza para crear, madurar y optimizar uno o más entornos de nube. Para obtener más información, consulte Creación de su modelo operativo de nube.

etapas de adopción de la nube

Las siguientes son las cuatro fases por las que suelen pasar las empresas cuando migran a la Nube de AWS:

- Proyecto: ejecución de algunos proyectos relacionados con la nube con fines de prueba de concepto y aprendizaje
- Fundamento: realización de inversiones fundamentales para escalar la adopción de la nube (p. ej., crear una zona de aterrizaje, definir un CCoE, establecer un modelo de operaciones)

- Migración: migración de aplicaciones individuales
- Reinvención: optimización de productos y servicios e innovación en la nube

Stephen Orban definió estas etapas en la publicación The Journey Toward Cloud-First & the Stages of Adoption del blog de estrategia empresarial de la Nube de AWS. Para obtener información sobre cómo se relacionan con la estrategia de migración de AWS, consulte la Guía de preparación para la migración.

CMDB

Consulte base de datos de administración de configuración.

repositorio de código

Una ubicación donde el código fuente y otros activos, como documentación, muestras y scripts, se almacenan y actualizan mediante procesos de control de versiones. Algunos repositorios en la nube comunes son GitHub o Bitbucket Cloud. Cada versión del código se denomina rama. En una estructura de microservicios, cada repositorio se encuentra dedicado a una única funcionalidad. Una sola canalización de CI/CD puede utilizar varios repositorios.

caché en frío

Una caché de búfer que está vacía no está bien poblada o contiene datos obsoletos o irrelevantes. Esto afecta al rendimiento, ya que la instancia de la base de datos debe leer desde la memoria principal o el disco, lo que es más lento que leer desde la memoria caché del búfer.

datos fríos

Datos a los que se accede con poca frecuencia y que suelen ser históricos. Al consultar este tipo de datos, normalmente se aceptan consultas lentas. Trasladar estos datos a niveles o clases de almacenamiento de menor rendimiento y menos costosos puede reducir los costos.

visión artificial (CV)

Campo de la IA que utiliza el machine learning para analizar y extraer información de formatos visuales, como imágenes y videos digitales. Por ejemplo, Amazon SageMaker AI proporciona algoritmos de procesamiento de imágenes para CV.

deriva de configuración

En el caso de una carga de trabajo, un cambio en la configuración con respecto al estado esperado. Podría provocar que la carga de trabajo deje de cumplir las normas y, por lo general, es gradual e involuntaria.

base de datos de administración de configuración (CMDB)

Repositorio que almacena y administra información sobre una base de datos y su entorno de TI, incluidos los componentes de hardware y software y sus configuraciones. Por lo general, los datos de una CMDB se utilizan en la etapa de detección y análisis de la cartera de productos durante la migración.

paquete de conformidad

Una colección de acciones correctivas y reglas de AWS Config que puede reunir para personalizar sus controles de seguridad y conformidad. Puede implementar un paquete de conformidad como una sola entidad en una región y Cuenta de AWS, o en toda una organización, mediante una plantilla YAML. Para obtener más información, consulte Paquetes de conformidad en la documentación de AWS Config.

integración y entrega continuas (CI/CD)

El proceso de automatización de las etapas de origen, compilación, prueba, presentación y producción del proceso de lanzamiento del software. La CI/CD se describe comúnmente como una canalización. La CI/CD puede ayudarlo a automatizar los procesos, mejorar la productividad, mejorar la calidad del código y entregar con mayor rapidez. Para obtener más información, consulte Beneficios de la entrega continua. CD también puede significar implementación continua. Para obtener más información, consulte Entrega continua frente a implementación continua.

CV

Consulte visión artificial.

D

datos en reposo

Datos que están estacionarios en la red, como los datos que se encuentran almacenados.

clasificación de datos

Un proceso para identificar y clasificar los datos de su red en función de su importancia y sensibilidad. Es un componente fundamental de cualquier estrategia de administración de riesgos de ciberseguridad porque lo ayuda a determinar los controles de protección y retención adecuados para los datos. La clasificación de datos es un componente del pilar de seguridad del Marco de AWS Well-Architected. Para obtener más información, consulte Clasificación de datos.

desviación de datos

Una variación significativa entre los datos de producción y los datos que se utilizaron para entrenar un modelo de machine learning, o un cambio significativo en los datos de entrada a lo largo del tiempo. La desviación de los datos puede reducir la calidad, la precisión y la imparcialidad generales de las predicciones de los modelos de machine learning.

datos en tránsito

Datos que se mueven de forma activa por la red, por ejemplo, entre los recursos de la red.

malla de datos

Marco de arquitectura que proporciona una propiedad de datos distribuida y descentralizada con una administración y una gobernanza centralizadas.

minimización de datos

El principio de recopilar y procesar solo los datos estrictamente necesarios. Practicar la minimización de los datos en Nube de AWS puede reducir los riesgos de privacidad, los costos y la huella de carbono derivada de los análisis.

perímetro de datos

Conjunto de barreras de protección en su entorno de AWS que ayudan a garantizar que solo las identidades de confianza accedan a los recursos de confianza desde las redes esperadas. Para más información, consulte Building a data perimeter on AWS.

preprocesamiento de datos

Transformar los datos sin procesar en un formato que su modelo de ML pueda analizar fácilmente. El preprocesamiento de datos puede implicar eliminar determinadas columnas o filas y corregir los valores faltantes, incoherentes o duplicados.

procedencia de los datos

El proceso de rastrear el origen y el historial de los datos a lo largo de su ciclo de vida, por ejemplo, la forma en que se generaron, transmitieron y almacenaron los datos.

titular de los datos

Persona cuyos datos se recopilan y procesan.

almacenamiento de datos

Sistema de administración de datos que respalda la inteligencia empresarial, como los análisis. Los almacenes de datos suelen contener grandes cantidades de datos históricos y, por lo general, se utilizan para las consultas y los análisis.

lenguaje de definición de datos (DDL)

Instrucciones o comandos para crear o modificar la estructura de tablas y objetos de una base de datos.

lenguaje de manipulación de datos (DML)

Instrucciones o comandos para modificar (insertar, actualizar y eliminar) la información de una base de datos.

DDL

Consulte lenguaje de definición de bases de datos.

conjunto profundo

Combinar varios modelos de aprendizaje profundo para la predicción. Puede utilizar conjuntos profundos para obtener una predicción más precisa o para estimar la incertidumbre de las predicciones.

aprendizaje profundo

Un subcampo del ML que utiliza múltiples capas de redes neuronales artificiales para identificar el mapeo entre los datos de entrada y las variables objetivo de interés.

defensa en profundidad

Un enfoque de seguridad de la información en el que se distribuyen cuidadosamente una serie de mecanismos y controles de seguridad en una red informática para proteger la confidencialidad, la integridad y la disponibilidad de la red y de los datos que contiene. Cuando se adopta esta estrategia en AWS, se suman varios controles en diferentes capas de la estructura de AWS Organizations para ayudar a proteger los recursos. Por ejemplo, un enfoque de defensa en profundidad podría combinar la autenticación multifactor, la segmentación de la red y el cifrado.

administrador delegado

En AWS Organizations, un servicio compatible puede registrar una cuenta de miembro de AWS a fin de administrar las cuentas de la organización y los permisos para ese servicio. Esta cuenta

se denomina administrador delegado para ese servicio. Para obtener más información y una lista de servicios compatibles, consulte Servicios que funcionan con AWS Organizations en la documentación de AWS Organizations.

implementación

El proceso de hacer que una aplicación, características nuevas o correcciones de código se encuentren disponibles en el entorno de destino. La implementación abarca implementar cambios en una base de código y, a continuación, crear y ejecutar esa base en los entornos de la aplicación.

entorno de desarrollo

Consulte entorno.

control de detección

Un control de seguridad que se ha diseñado para detectar, registrar y alertar después de que se produzca un evento. Estos controles son una segunda línea de defensa, ya que lo advierten sobre los eventos de seguridad que han eludido los controles preventivos establecidos. Para obtener más información, consulte Controles de detección en Implementación de controles de seguridad en AWS.

asignación de flujos de valor para el desarrollo (DVSM)

Proceso que se utiliza para identificar y priorizar las restricciones que afectan negativamente a la velocidad y la calidad en el ciclo de vida del desarrollo de software. DVSM amplía el proceso de asignación del flujo de valor diseñado originalmente para las prácticas de fabricación ajustada. Se centra en los pasos y los equipos necesarios para crear y transferir valor a través del proceso de desarrollo de software.

gemelo digital

Representación virtual de un sistema del mundo real, como un edificio, una fábrica, un equipo industrial o una línea de producción. Los gemelos digitales son compatibles con el mantenimiento predictivo, la supervisión remota y la optimización de la producción.

tabla de dimensiones

En un esquema en estrella, tabla más pequeña que contiene los atributos de datos sobre los datos cuantitativos en una tabla de hechos. Los atributos de la tabla de dimensiones suelen ser campos de texto o números discretos que se comportan como texto. Estos atributos se suelen utilizar para restringir consultas, filtrarlas y etiquetar los conjuntos de resultados.

desastre

Un evento que impide que una carga de trabajo o un sistema cumplan sus objetivos empresariales en su ubicación principal de implementación. Estos eventos pueden ser desastres naturales, fallos técnicos o el resultado de acciones humanas, como una configuración incorrecta involuntaria o un ataque de malware.

recuperación de desastres (DR)

Estrategia y proceso que utiliza para minimizar el tiempo de inactividad y la pérdida de datos a causa de un desastre. Para obtener más información, consulte Recuperación de desastres de cargas de trabajo en AWS: Recuperación en la nube en un marco Well-Architected de AWS.

DML

Consulte lenguaje de manipulación de bases de datos.

diseño basado en el dominio

Un enfoque para desarrollar un sistema de software complejo mediante la conexión de sus componentes a dominios en evolución, o a los objetivos empresariales principales, a los que sirve cada componente. Este concepto lo introdujo Eric Evans en su libro, Diseño impulsado por el dominio: abordando la complejidad en el corazón del software (Boston: Addison-Wesley Professional, 2003). Para obtener información sobre cómo utilizar el diseño basado en dominios con el patrón de higos estranguladores, consulte Modernización gradual de los servicios web antiguos de Microsoft ASP.NET (ASMX) mediante contenedores y Amazon API Gateway.

DR

Consulte recuperación ante desastres.

detección de desviaciones

Seguimiento de las desviaciones con respecto a una configuración con línea de base. Por ejemplo, puede usar AWS CloudFormation para detectar desviaciones en los recursos del sistema o puede usar AWS Control Tower para detectar cambios en la zona de aterrizaje que podrían afectar al cumplimiento de los requisitos de gobernanza.

DVSM

Consulte asignación de flujos de valor para el desarrollo.

E

EDA

Consulte análisis de datos de tipo exploratorio.

EDI

Consulte intercambio electrónico de datos.

computación en la periferia

La tecnología que aumenta la potencia de cálculo de los dispositivos inteligentes en la periferia de una red de IoT. En comparación con la computación en la nube, la computación de periferia puede reducir la latencia de la comunicación y mejorar el tiempo de respuesta.

intercambio electrónico de datos (EDI)

Intercambio automatizado de documentos comerciales entre organizaciones. Para más información, consulte ¿Qué es el intercambio electrónico de datos?

cifrado

Proceso de computación que transforma datos de texto plano, que son legibles por humanos, en texto cifrado.

clave de cifrado

Cadena criptográfica de bits aleatorios que se genera mediante un algoritmo de cifrado. Las claves pueden variar en longitud y cada una se ha diseñado para ser impredecible y única.

endianidad

El orden en el que se almacenan los bytes en la memoria del ordenador. Los sistemas big-endianos almacenan primero el byte más significativo. Los sistemas Little-Endian almacenan primero el byte menos significativo.

endpoint

Consulte punto de conexión de servicio.

servicio de punto de conexión

Servicio que puede alojar en una nube privada virtual (VPC) para compartir con otros usuarios. Puede crear un servicio de punto de conexión con AWS PrivateLink y conceder permisos a otras Cuentas de AWS o para entidades principales de AWS Identity and Access Management (IAM).

Estas cuentas o entidades principales pueden conectarse a su servicio de punto de conexión de forma privada mediante la creación de puntos de conexión de VPC de interfaz. Para obtener más información, consulte Creación de un servicio de punto de conexión en la documentación de Amazon Virtual Private Cloud (Amazon VPC).

planificación de recursos empresariales (ERP)

Sistema que automatiza y administra los procesos empresariales clave (como la contabilidad, MES y la administración de proyectos) de una empresa.

cifrado de sobre

El proceso de cifrar una clave de cifrado con otra clave de cifrado. Para obtener más información, consulte Cifrado de sobre en la documentación de AWS Key Management Service (AWS KMS).

entorno

Una instancia de una aplicación en ejecución. Los siguientes son los tipos de entornos más comunes en la computación en la nube:

- entorno de desarrollo: instancia de una aplicación en ejecución que solo se encuentra disponible para el equipo principal responsable del mantenimiento de la aplicación. Los entornos de desarrollo se utilizan para probar los cambios antes de promocionarlos a los entornos superiores. Este tipo de entorno a veces se denomina entorno de prueba.
- entornos inferiores: todos los entornos de desarrollo de una aplicación, como los que se utilizan para las compilaciones y pruebas iniciales.
- entorno de producción: instancia de una aplicación en ejecución a la que pueden acceder los usuarios finales. En una canalización de CI/CD, el entorno de producción es el último entorno de implementación.
- entornos superiores: todos los entornos a los que pueden acceder usuarios que no sean del equipo de desarrollo principal. Esto puede incluir un entorno de producción, entornos de preproducción y entornos para las pruebas de aceptación por parte de los usuarios.

epopeya

En las metodologías ágiles, son categorías funcionales que ayudan a organizar y priorizar el trabajo. Las epopeyas brindan una descripción detallada de los requisitos y las tareas de implementación. Por ejemplo, las epopeyas de seguridad de AWS CAF incluyen la administración de identidades y accesos, los controles de detección, la seguridad de la infraestructura, la protección de datos y la respuesta a incidentes. Para obtener más información sobre las epopeyas en la estrategia de migración de AWS, consulte la Guía de implementación del programa.

ERP

Consulte planificación de recursos empresariales.

análisis de datos de tipo exploratorio (EDA)

El proceso de analizar un conjunto de datos para comprender sus características principales. Se recopilan o agregan datos y, a continuación, se realizan las investigaciones iniciales para encontrar patrones, detectar anomalías y comprobar las suposiciones. El EDA se realiza mediante el cálculo de estadísticas resumidas y la creación de visualizaciones de datos.

F

tabla de hechos

Tabla central de un esquema en estrella. Almacena datos cuantitativos sobre operaciones empresariales. Por lo general, una tabla de hechos contiene dos tipos de columnas: las que contienen medidas y las que contienen una clave externa para una tabla de dimensiones.

respuesta rápida a los errores

Filosofía que utiliza pruebas frecuentes e incrementales para reducir el ciclo de vida del desarrollo. Es una parte fundamental de los enfoques ágiles.

límite de aislamiento de errores

En la Nube de AWS, límite, como una zona de disponibilidad, una Región de AWS, un plano de control o un plano de datos, que acota el efecto de un error y ayuda a mejorar la resiliencia de las cargas de trabajo. Para más información, consulte AWS Fault Isolation Boundaries.

rama de característica

Consulte rama.

características

Los datos de entrada que se utilizan para hacer una predicción. Por ejemplo, en un contexto de fabricación, las características pueden ser imágenes que se capturan periódicamente desde la línea de fabricación.

importancia de las características

La importancia que tiene una característica para las predicciones de un modelo. Por lo general, esto se expresa como una puntuación numérica que se puede calcular mediante diversas

técnicas, como las explicaciones aditivas de Shapley (SHAP) y los gradientes integrados. Para más información, consulte Machine learning model interpretability with AWS.

transformación de funciones

Optimizar los datos para el proceso de ML, lo que incluye enriquecer los datos con fuentes adicionales, escalar los valores o extraer varios conjuntos de información de un solo campo de datos. Esto permite que el modelo de ML se beneficie de los datos. Por ejemplo, si divide la fecha del “27 de mayo de 2021 00:15:37” en “jueves”, “mayo”, “2021” y “15”, puede ayudar al algoritmo de aprendizaje a aprender patrones matizados asociados a los diferentes componentes de los datos.

peticiones con pocos pasos

Proporcionar a un LLM una pequeña cantidad de ejemplos que demuestren la tarea y el resultado deseado antes de pedirle que lleve a cabo una tarea similar. Esta técnica es una aplicación del aprendizaje contextual, mediante el que los modelos aprenden a partir de ejemplos (pasos) incrustados en las peticiones. La técnica de peticiones con pocos pasos puede ser eficaz para las tareas que requieren un formato, un razonamiento o un conocimiento del dominio específicos. Consulte también peticiones desde cero.

FGAC

Consulte control de acceso detallado.

control de acceso preciso (FGAC)

El uso de varias condiciones que tienen por objetivo permitir o denegar una solicitud de acceso.

migración relámpago

Método de migración de bases de datos que utiliza la replicación continua de datos mediante la captura de datos de cambio para migrar los datos en el menor tiempo posible, en lugar de utilizar un enfoque gradual. El objetivo es reducir al mínimo el tiempo de inactividad.

FM

Consulte modelo fundacional.

modelo fundacional (FM)

Gran red neuronal de aprendizaje profundo que se entrenó con conjuntos de datos masivos de datos generalizados y no etiquetados. Los FM pueden hacer una amplia variedad de tareas generales, como comprender el lenguaje, generar texto e imágenes y conversar en lenguaje natural. Para más información, consulte ¿Qué son los modelos fundacionales?

G

IA generativa

Subconjunto de modelos de IA que se entrenaron con grandes cantidades de datos y que pueden utilizar una simple petición de texto para crear contenido y artefactos nuevos, como imágenes, videos, texto y audio. Para más información, consulte ¿Qué es la IA generativa?

bloqueo geográfico

Consulte restricciones geográficas.

restricciones geográficas (bloqueo geográfico)

En Amazon CloudFront, una opción para impedir que los usuarios de países específicos accedan a las distribuciones de contenido. Puede utilizar una lista de permitidos o bloqueados para especificar los países aprobados y prohibidos. Para obtener más información, consulte Restricción de la distribución geográfica de su contenido en la documentación de CloudFront.

Flujo de trabajo de Gitflow

Un enfoque en el que los entornos inferiores y superiores utilizan diferentes ramas en un repositorio de código fuente. El flujo de trabajo de Gitflow se considera heredado, mientras que el flujo de trabajo basado en enlaces troncales es el enfoque moderno preferido.

imagen dorada

Instantánea de un sistema o software que se usa como plantilla para implementar nuevas instancias de ese sistema o software. Por ejemplo, en la fabricación, una imagen dorada se puede utilizar para aprovisionar software en varios dispositivos y ayuda a mejorar la velocidad, la escalabilidad y la productividad de las operaciones de fabricación de dispositivos.

estrategia de implementación desde cero

La ausencia de infraestructura existente en un entorno nuevo. Al adoptar una estrategia de implementación desde cero para una arquitectura de sistemas, puede seleccionar todas las tecnologías nuevas sin que estas deban ser compatibles con una infraestructura existente, lo que también se conoce como implementación sobre infraestructura existente. Si está ampliando la infraestructura existente, puede combinar las estrategias de implementación sobre infraestructuras existentes y de implementación desde cero.

barrera de protección

Una regla de alto nivel que ayuda a regular los recursos, las políticas y la conformidad en todas las unidades organizativas (OU). Las barreras de protección preventivas aplican políticas para garantizar la alineación con los estándares de conformidad. Se implementan mediante políticas de control de servicios y límites de permisos de IAM. Las barreras de protección de detección detectan las vulneraciones de las políticas y los problemas de conformidad, y generan alertas para su corrección. Se implementan mediante el uso de AWS Config, AWS Security Hub CSPM, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector y comprobaciones de AWS Lambda personalizadas.

H

HA

Consulte alta disponibilidad.

migración heterogénea de bases de datos

Migración de la base de datos de origen a una base de datos de destino que utilice un motor de base de datos diferente (por ejemplo, de Oracle a Amazon Aurora). La migración heterogénea suele ser parte de un esfuerzo de rediseño de la arquitectura y convertir el esquema puede ser una tarea compleja. AWS ofrece AWS SCT, lo cual ayuda con las conversiones de esquemas.

alta disponibilidad (HA)

La capacidad de una carga de trabajo para funcionar de forma continua, sin intervención, en caso de desafíos o desastres. Los sistemas de alta disponibilidad están diseñados para realizar una conmutación por error automática, ofrecer un rendimiento de alta calidad de forma constante y gestionar diferentes cargas y fallos con un impacto mínimo en el rendimiento.

modernización histórica

Un enfoque utilizado para modernizar y actualizar los sistemas de tecnología operativa (TO) a fin de satisfacer mejor las necesidades de la industria manufacturera. Un histórico es un tipo de base de datos que se utiliza para recopilar y almacenar datos de diversas fuentes en una fábrica.

datos de reserva

Parte de los datos históricos etiquetados que se ocultan de un conjunto de datos que se utiliza para entrenar un modelo de machine learning. Puede utilizar los datos de reserva para evaluar el

rendimiento del modelo mediante la comparación de las predicciones del modelo con los datos de reserva.

migración homogénea de bases de datos

Migración de la base de datos de origen a una base de datos de destino que comparte el mismo motor de base de datos (por ejemplo, Microsoft SQL Server a Amazon RDS para SQL Server). La migración homogénea suele formar parte de un esfuerzo para volver a alojar o redefinir la plataforma. Puede utilizar las utilidades de bases de datos nativas para migrar el esquema.

datos recientes

Datos a los que se accede con frecuencia, como datos en tiempo real o datos traslacionales recientes. Por lo general, estos datos requieren un nivel o una clase de almacenamiento de alto rendimiento para proporcionar respuestas rápidas a las consultas.

hotfix

Una solución urgente para un problema crítico en un entorno de producción. Debido a su urgencia, el hotfix suele realizarse fuera del flujo de trabajo típico de las versiones de DevOps.

periodo de hiperatención

Periodo, inmediatamente después de la transición, durante el cual un equipo de migración administra y monitorea las aplicaciones migradas en la nube para solucionar cualquier problema. Por lo general, este periodo dura de 1 a 4 días. Al final del periodo de hiperatención, el equipo de migración suele transferir la responsabilidad de las aplicaciones al equipo de operaciones en la nube.

I

IaC

Consulte infraestructura como código.

políticas basadas en identidad

Una política asociada a una o más entidades principales de IAM que define sus permisos en el entorno de la Nube de AWS.

aplicación inactiva

Aplicación que utiliza un promedio de CPU y memoria de entre 5 y 20 por ciento durante un periodo de 90 días. En un proyecto de migración, es habitual retirar estas aplicaciones o mantenerlas en las instalaciones.

IIoT

Consulte Internet de las cosas industrial.

infraestructura inmutable

Modelo que implementa una nueva infraestructura para las cargas de trabajo de producción en lugar de actualizar o modificar la infraestructura existente o aplicarle revisiones. Las infraestructuras inmutables son intrínsecamente más coherentes, fiables y predecibles que las infraestructuras mutables. Para más información, consulte la práctica recomendada Implementación mediante una infraestructura inmutable en el Marco de AWS Well-Architected.

VPC entrante (de entrada)

En una arquitectura de varias cuentas de AWS, una VPC que acepta, inspecciona y enruta las conexiones de red desde fuera de una aplicación. La Arquitectura de referencia de seguridad de AWS recomienda configurar su cuenta de red con VPC entrantes, salientes y de inspección para proteger la interfaz bidireccional entre su aplicación e Internet en general.

migración gradual

Estrategia de transición en la que se migra la aplicación en partes pequeñas en lugar de realizar una transición única y completa. Por ejemplo, puede trasladar inicialmente solo unos pocos microservicios o usuarios al nuevo sistema. Tras comprobar que todo funciona correctamente, puede trasladar microservicios o usuarios adicionales de forma gradual hasta que pueda retirar su sistema heredado. Esta estrategia reduce los riesgos asociados a las grandes migraciones.

Industria 4.0

Término que introdujo Klaus Schwab en 2016 para referirse a la modernización de los procesos de fabricación mediante los avances en la conectividad, los datos en tiempo real, la automatización, el análisis, la IA y el ML.

infraestructura

Todos los recursos y activos que se encuentran en el entorno de una aplicación.

infraestructura como código (IaC)

Proceso de aprovisionamiento y administración de la infraestructura de una aplicación mediante un conjunto de archivos de configuración. La IaC se ha diseñado para ayudarlo a centralizar la administración de la infraestructura, estandarizar los recursos y escalar con rapidez a fin de que los entornos nuevos sean repetibles, fiables y consistentes.

Internet de las cosas industrial (IIoT)

El uso de sensores y dispositivos conectados a Internet en los sectores industriales, como el productivo, el eléctrico, el automotriz, el sanitario, el de las ciencias de la vida y el de la agricultura. Para obtener más información, consulte Creación de una estrategia de transformación digital del Internet de las cosas industrial (IIoT).

VPC de inspección

En una arquitectura de varias cuentas de AWS, una VPC centralizada que administra las inspecciones del tráfico de red entre VPC (en la misma o en diferentes Regiones de AWS), Internet y las redes en las instalaciones. La Arquitectura de referencia de seguridad de AWS recomienda configurar su cuenta de red con VPC entrantes, salientes y de inspección para proteger la interfaz bidireccional entre su aplicación e Internet en general.

Internet de las cosas (IoT)

Red de objetos físicos conectados con sensores o procesadores integrados que se comunican con otros dispositivos y sistemas a través de Internet o de una red de comunicación local. Para obtener más información, consulte ¿Qué es IoT?.

interpretabilidad

Característica de un modelo de machine learning que describe el grado en que un ser humano puede entender cómo las predicciones del modelo dependen de sus entradas. Para más información, consulte Machine learning model interpretability with AWS.

IoT

Consulte Internet de las cosas.

biblioteca de información de TI (ITIL)

Conjunto de prácticas recomendadas para ofrecer servicios de TI y alinearlos con los requisitos empresariales. La ITIL proporciona la base para la ITSM.

administración de servicios de TI (ITSM)

Actividades asociadas con el diseño, la implementación, la administración y el soporte de los servicios de TI para una organización. Para obtener información sobre la integración de las operaciones en la nube con las herramientas de ITSM, consulte la Guía de integración de operaciones.

ITIL

Consulte biblioteca de información de TI.

ITSM

Consulte administración de servicios de TI.

L

control de acceso basado en etiquetas (LBAC)

Una implementación del control de acceso obligatorio (MAC) en la que a los usuarios y a los propios datos se les asigna explícitamente un valor de etiqueta de seguridad. La intersección entre la etiqueta de seguridad del usuario y la etiqueta de seguridad de los datos determina qué filas y columnas puede ver el usuario.

zona de aterrizaje

Una zona de aterrizaje es un entorno de AWS correctamente diseñado, con varias cuentas, que es escalable y seguro. Este es un punto de partida desde el cual las empresas pueden lanzar e implementar rápidamente cargas de trabajo y aplicaciones con confianza en su entorno de seguridad e infraestructura. Para obtener más información sobre las zonas de aterrizaje, consulte Configuración de un entorno de AWS seguro y escalable con varias cuentas.

modelo de lenguaje de gran tamaño (LLM)

Modelo de IA de aprendizaje profundo que se entrenó previamente con una gran cantidad de datos. Un LLM puede llevar a cabo varias tareas, como responder preguntas, resumir documentos, traducir textos a otros idiomas y completar oraciones. Para más información, consulte ¿Qué es un LLM (modelo de lenguaje de gran tamaño)?

migración grande

Migración de 300 servidores o más.

LBAC

Consulte control de acceso basado en etiquetas.

privilegio mínimo

La práctica recomendada de seguridad que consiste en conceder los permisos mínimos necesarios para realizar una tarea. Para obtener más información, consulte Aplicar permisos de privilegio mínimo en la documentación de IAM.

migrar mediante lift-and-shift

Consulte Las 7 R.

sistema little-endian

Un sistema que almacena primero el byte menos significativo. Consulte también endianidad.

LLM

Consulte modelo de lenguaje de gran tamaño.

entornos inferiores

Consulte entorno.

M

machine learning (ML)

Un tipo de inteligencia artificial que utiliza algoritmos y técnicas para el reconocimiento y el aprendizaje de patrones. El ML analiza y aprende de los datos registrados, como los datos del Internet de las cosas (IoT), para generar un modelo estadístico basado en patrones. Para más información, consulte Machine learning.

rama principal

Consulte rama.

malware

Software diseñado para comprometer la seguridad o la privacidad de la computadora. El malware podría interrumpir los sistemas informáticos, filtrar información confidencial u obtener acceso no autorizado. Algunos ejemplos de malware son los virus, los gusanos, el ransomware, los troyanos, el spyware y los registradores de pulsaciones de teclas.

servicios administrados

Servicios de AWS para los que AWS gestiona la capa de infraestructura, el sistema operativo y las plataformas, mientras que se accede a los puntos de conexión para almacenar y recuperar datos. Amazon Simple Storage Service (Amazon S3) y Amazon DynamoDB son ejemplos de servicios administrados. También se conocen como servicios abstractos.

sistema de ejecución de fabricación (MES)

Sistema de software para seguir, supervisar, documentar y controlar los procesos de producción que convierten las materias primas en productos acabados en la zona de producción.

MAP

Consulte Programa de aceleración de la migración.

mecanismo

Proceso completo mediante el que se crea una herramienta, se impulsa su adopción y, a continuación, se inspeccionan los resultados para hacer ajustes. Un mecanismo es un ciclo que se refuerza y mejora por sí mismo a medida que funciona. Para más información, consulte Building mechanisms en el Marco de AWS Well-Architected.

cuenta de miembro

Todas las Cuentas de AWS distintas de las cuentas de administración que forman parte de una organización en AWS Organizations. Una cuenta no puede pertenecer a más de una organización a la vez.

MES

Consulte sistema de ejecución de fabricación.

Message Queuing Telemetry Transport (MQTT)

Protocolo de comunicación ligero de máquina a máquina (M2M) que se basa en el patrón de publicación/suscripción y está pensado para dispositivos de IoT con recursos limitados.

microservicio

Un servicio pequeño e independiente que se comunica a través de API bien definidas y que, por lo general, es propiedad de equipos pequeños e independientes. Por ejemplo, un sistema de seguros puede incluir microservicios que se adapten a las capacidades empresariales, como las de ventas o marketing, o a subdominios, como las de compras, reclamaciones o análisis. Los beneficios de los microservicios incluyen la agilidad, la escalabilidad flexible, la facilidad de

implementación, el código reutilizable y la resiliencia. Para obtener más información, consulte Integración de microservicios mediante servicios sin servidor de AWS.

arquitectura de microservicios

Un enfoque para crear una aplicación con componentes independientes que ejecutan cada proceso de la aplicación como un microservicio. Estos microservicios se comunican a través de una interfaz bien definida mediante API ligeras. Cada microservicio de esta arquitectura se puede actualizar, implementar y escalar para satisfacer la demanda de funciones específicas de una aplicación. Para obtener más información, consulte Implementación de microservicios en AWS.

Programa de aceleración de la migración (MAP)

Programa de AWS que brinda soporte de consultoría, capacitación y servicios para ayudar a las empresas a construir una base operativa sólida para migrar a la nube y ayudar a compensar el costo inicial de las migraciones. El MAP incluye una metodología de migración para ejecutar las migraciones antiguas de forma metódica y un conjunto de herramientas para automatizar y acelerar los escenarios de migración más comunes.

migración a escala

Proceso de transferencia de la mayoría de la cartera de aplicaciones a la nube en oleadas, con más aplicaciones desplazadas a un ritmo más rápido en cada oleada. En esta fase, se utilizan las prácticas recomendadas y las lecciones aprendidas en las fases anteriores para implementar una fábrica de migración de equipos, herramientas y procesos con el fin de agilizar la migración de las cargas de trabajo mediante la automatización y la entrega ágil. Esta es la tercera fase de la estrategia de migración de AWS.

fábrica de migración

Equipos multifuncionales que agilizan la migración de las cargas de trabajo mediante enfoques automatizados y ágiles. Los equipos de la fábrica de migración suelen incluir operaciones, analistas y propietarios de negocios, ingenieros de migración, desarrolladores y profesionales de DevOps que trabajan en tiempo y forma. Entre el 20 y el 50 por ciento de la cartera de aplicaciones empresariales se compone de patrones repetidos que pueden optimizarse mediante un enfoque de fábrica. Para obtener más información, consulte la discusión sobre las fábricas de migración y la Guía de fábricas de migración a la nube en este contenido.

metadatos de migración

Información sobre la aplicación y el servidor que se necesita para completar la migración. Cada patrón de migración requiere un conjunto diferente de metadatos de migración. Algunos ejemplos

de metadatos de migración son las subredes de destino, los grupos de seguridad y las cuentas de AWS.

patrón de migración

Tarea de migración repetible que detalla la estrategia de migración, el destino de la migración y la aplicación o el servicio de migración utilizados. Ejemplo: Volver a alojar la migración en Amazon EC2 con AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

Herramienta en línea que proporciona información a fin de validar los argumentos comerciales necesarios para migrar a la Nube de AWS. La MPA ofrece una evaluación detallada de la cartera (adecuación del tamaño de los servidores, precios, comparaciones del costo total de propiedad, análisis de los costos de migración), así como una planificación de la migración (análisis y recopilación de datos de aplicaciones, agrupación de aplicaciones, priorización de la migración y planificación de oleadas). La herramienta MPA (requiere inicio de sesión) está disponible de forma gratuita para todos los consultores de AWS y los consultores asociados de APN.

Evaluación de la preparación para la migración (MRA)

Proceso que consiste en obtener información sobre el estado de preparación de la nube de una organización, identificar los puntos fuertes y débiles, y elaborar un plan de acción para cerrar las brechas identificadas, mediante AWS CAF. Para obtener más información, consulte la Guía de preparación para la migración. La MRA es la primera fase de la estrategia de migración de AWS.

estrategia de migración

Enfoque utilizado para migrar una carga de trabajo a la Nube de AWS. Para más información, consulte la entrada Las 7 R de este glosario y también Mobilize your organization to accelerate large-scale migrations.

ML

Consulte machine learning.

modernización

Transformar una aplicación obsoleta (antigua o monolítica) y su infraestructura en un sistema ágil, elástico y de alta disponibilidad en la nube para reducir los gastos, aumentar la eficiencia y aprovechar las innovaciones. Para más información, consulte Strategy for modernizing applications in the Nube de AWS.

evaluación de la preparación para la modernización

Evaluación que ayuda a determinar la preparación para la modernización de las aplicaciones de una organización; identifica los beneficios, los riesgos y las dependencias; y determina qué tan bien la organización puede soportar el estado futuro de esas aplicaciones. El resultado de la evaluación es un esquema de la arquitectura objetivo, una hoja de ruta que detalla las fases de desarrollo y los hitos del proceso de modernización y un plan de acción para abordar las brechas identificadas. Para más información, consulte Evaluating modernization readiness for applications in the Nube de AWS.

aplicaciones monolíticas (monolitos)

Aplicaciones que se ejecutan como un único servicio con procesos estrechamente acoplados. Las aplicaciones monolíticas presentan varios inconvenientes. Si una característica de la aplicación experimenta un aumento en la demanda, se debe escalar toda la arquitectura. Agregar o mejorar las características de una aplicación monolítica también se vuelve más complejo a medida que crece la base de código. Para solucionar problemas con la aplicación, puede utilizar una arquitectura de microservicios. Para obtener más información, consulte Descomposición de monolitos en microservicios.

MPA

Consulte Migration Portfolio Assessment.

MQTT

Consulte Message Queuing Telemetry Transport.

clasificación multiclase

Un proceso que ayuda a generar predicciones para varias clases (predice uno de más de dos resultados). Por ejemplo, un modelo de ML podría preguntar “¿Este producto es un libro, un automóvil o un teléfono?” o “¿Qué categoría de productos es más interesante para este cliente?”.

infraestructura mutable

Modelo que actualiza y modifica la infraestructura actual para las cargas de trabajo de producción. Para mejorar la coherencia, la fiabilidad y la previsibilidad, el Marco de AWS Well-Architected recomienda usar una infraestructura inmutable.

O

OAC

Consulte control de acceso de origen.

OAI

Consulte identidad de acceso de origen.

OCM

Consulte administración del cambio organizacional.

migración fuera de línea

Método de migración en el que la carga de trabajo de origen se elimina durante el proceso de migración. Este método implica un tiempo de inactividad prolongado y, por lo general, se utiliza para cargas de trabajo pequeñas y no críticas.

OI

Consulte integración de operaciones.

OLA

Consulte acuerdo de nivel operativo.

migración en línea

Método de migración en el que la carga de trabajo de origen se copia al sistema de destino sin que se desconecte. Las aplicaciones que están conectadas a la carga de trabajo pueden seguir funcionando durante la migración. Este método implica un tiempo de inactividad nulo o mínimo y, por lo general, se utiliza para cargas de trabajo de producción críticas.

OPC-UA

Consulte Open Process Communications: arquitectura unificada.

Open Process Communications: arquitectura unificada (OPC-UA)

Protocolo de comunicación de máquina a máquina (M2M) para la automatización industrial. OPC-UA establece un estándar de interoperabilidad con esquemas de autenticación, autorización y cifrado de datos.

acuerdo de nivel operativo (OLA)

Acuerdo que aclara lo que los grupos de TI operativos se comprometen a ofrecerse entre sí, para respaldar un acuerdo de nivel de servicio (SLA).

revisión de la preparación operativa (ORR)

Lista de comprobación de preguntas y prácticas recomendadas asociadas que son útiles para comprender, evaluar, prevenir o reducir el alcance de los incidentes y posibles errores. Para más información, consulte Operational Readiness Reviews (ORR) en el Marco de AWS Well-Architected.

tecnología operativa (TO)

Sistemas de hardware y software que funcionan con el entorno físico para controlar las operaciones, los equipos y la infraestructura industriales. En el sector de la fabricación, la integración de los sistemas de TO y tecnología de la información (TI) es un enfoque clave para las transformaciones de la industria 4.0.

integración de operaciones (OI)

Proceso de modernización de las operaciones en la nube, que implica la planificación de la preparación, la automatización y la integración. Para obtener más información, consulte la Guía de integración de las operaciones.

registro de seguimiento organizativo

Registro de seguimiento creado por AWS CloudTrail que registra todos los eventos para todas las Cuentas de AWS en una organización en AWS Organizations. Este registro de seguimiento se crea en cada Cuenta de AWS que forma parte de la organización y realiza un seguimiento de la actividad en cada cuenta. Para obtener más información, consulte Creación de un registro de seguimiento para una organización en la documentación de CloudTrail.

administración del cambio organizacional (OCM)

Marco para administrar las transformaciones empresariales importantes y disruptivas desde la perspectiva de las personas, la cultura y el liderazgo. La OCM ayuda a las empresas a prepararse para nuevos sistemas y estrategias y a realizar la transición a ellos, al acelerar la adopción de cambios, abordar los problemas de transición e impulsar cambios culturales y organizacionales. En la estrategia de migración de AWS, este marco se denomina aceleración de personas, debido a la velocidad de cambio requerida en los proyectos de adopción de la nube. Para obtener más información, consulte la Guía de OCM.

control de acceso de origen (OAC)

En CloudFront, una opción mejorada para restringir el acceso a su contenido de Amazon Simple Storage Service (Amazon S3). El OAC es compatible con todos los buckets de S3 en todas las Regiones de AWS, cifrado del servidor con AWS KMS (SSE-KMS), y solicitudes `PUT` y `DELETE` dinámicas al bucket de S3.

identidad de acceso de origen (OAI)

En CloudFront, una opción para restringir el acceso al contenido de Amazon S3. Cuando utiliza la OAI, CloudFront crea una entidad principal con la que Amazon S3 puede autenticarse. Las entidades principales autenticadas solo pueden acceder al contenido de un bucket de S3 a través de una distribución de CloudFront específica. Consulte también el OAC, que proporciona un control de acceso más detallado y mejorado.

ORR

Consulte revisión de la preparación operativa.

OT

Consulte tecnología operativa.

VPC saliente (de salida)

En una arquitectura de varias cuentas de AWS, una VPC que gestiona las conexiones de red que se inician desde una aplicación. La Arquitectura de referencia de seguridad de AWS recomienda configurar su cuenta de red con VPC entrantes, salientes y de inspección para proteger la interfaz bidireccional entre su aplicación e Internet en general.

P

límite de permisos

Una política de administración de IAM que se adjunta a las entidades principales de IAM para establecer los permisos máximos que puede tener el usuario o el rol. Para obtener más información, consulte Límites de permisos en la documentación de IAM.

información de identificación personal (PII)

Información que, vista directamente o combinada con otros datos relacionados, puede utilizarse para deducir de manera razonable la identidad de una persona. Algunos ejemplos de información de identificación personal son los nombres, las direcciones y la información de contacto.

PII

Consulte información de identificación personal.

manual de estrategias

Conjunto de pasos predefinidos que capturan el trabajo asociado a las migraciones, como la entrega de las funciones de operaciones principales en la nube. Un manual puede adoptar la forma de scripts, manuales de procedimientos automatizados o resúmenes de los procesos o pasos necesarios para operar un entorno modernizado.

PLC

Consulte controlador lógico programable.

PLM

Consulte administración del ciclo de vida del producto.

policy

Objeto que puede definir permisos (consulte política basada en identidad), especificar las condiciones de acceso (consulte política basada en recursos) o definir los permisos máximos para todas las cuentas de una organización de AWS Organizations (consulte política de control de servicio).

persistencia políglota

Elegir de forma independiente la tecnología de almacenamiento de datos de un microservicio en función de los patrones de acceso a los datos y otros requisitos. Si sus microservicios tienen la misma tecnología de almacenamiento de datos, pueden enfrentarse a desafíos de implementación o experimentar un rendimiento deficiente. Los microservicios se implementan más fácilmente y logran un mejor rendimiento y escalabilidad si utilizan el almacén de datos que mejor se adapte a sus necesidades. Para obtener más información, consulte Habilitación de la persistencia de datos en los microservicios.

evaluación de cartera

Proceso de detección, análisis y priorización de la cartera de aplicaciones para planificar la migración. Para obtener más información, consulte la Evaluación de la preparación para la migración.

predicate

Condición de consulta que devuelve `true` o `false`. En general, se encuentra en una cláusula `WHERE`.

inserción de predicados

Técnica de optimización de consultas en bases de datos que filtra los datos de la consulta antes de transferirlos. Esta técnica reduce la cantidad de datos de la base de datos relacional que se tienen que recuperar y procesar. Además, mejora el rendimiento de las consultas.

control preventivo

Un control de seguridad diseñado para evitar que ocurra un evento. Estos controles son la primera línea de defensa para evitar el acceso no autorizado o los cambios no deseados en la red. Para obtener más información, consulte Controles preventivos en Implementación de controles de seguridad en AWS.

entidad principal

Una entidad de AWS que puede realizar acciones y obtener acceso a los recursos. Esta entidad suele ser un usuario raíz de una Cuenta de AWS, un rol de IAM o un usuario. Para obtener más información, consulte Entidad principal en Términos y conceptos de roles en la documentación de IAM.

privacidad desde el diseño

Enfoque de ingeniería de sistemas que tiene en cuenta la privacidad durante todo el proceso de desarrollo.

zonas alojadas privadas

Contenedor que aloja información acerca de cómo desea que responda Amazon Route 53 a las consultas de DNS de un dominio y sus subdominios en una o varias VPC. Para obtener más información, consulte Uso de zonas alojadas privadas en la documentación de Route 53.

control proactivo

Control de seguridad que se diseñó para evitar la implementación de recursos que no cumplan con la normativa. Estos controles analizan los recursos antes de aprovisionarlos. Si el recurso no cumple con los requisitos del control, no se aprovisiona. Para más información, consulte Controls reference guide en la documentación de AWS Control Tower y consulte Proactive controls en Implementing security controls on AWS.

administración del ciclo de vida del producto (PLM)

Administración de los datos y los procesos de un producto a lo largo de todo su ciclo de vida, desde el diseño, el desarrollo y el lanzamiento, pasando por el crecimiento y la madurez, hasta la reducción de su uso y su retirada.

entorno de producción

Consulte entorno.

controlador lógico programable (PLC)

En el sector de la fabricación, computadora adaptable y altamente fiable que supervisa las máquinas y automatiza los procesos de fabricación.

encadenamiento de peticiones

Uso de la salida de una petición de LLM como entrada para la siguiente petición a fin de generar mejores respuestas. Esta técnica se utiliza para dividir una tarea compleja en tareas secundarias o para refinar o ampliar de forma iterativa una respuesta preliminar. Ayuda a mejorar la precisión y la relevancia de las respuestas de un modelo y permite obtener resultados más detallados y personalizados.

seudonimización

El proceso de reemplazar los identificadores personales de un conjunto de datos por valores de marcadores de posición. La seudonimización puede ayudar a proteger la privacidad personal. Los datos seudonimizados siguen considerándose datos personales.

publicación/suscripción (pub/sub)

Patrón que permite establecer comunicaciones asíncronas entre microservicios para mejorar la escalabilidad y la capacidad de respuesta. Por ejemplo, en un MES basado en microservicios, un microservicio puede publicar mensajes de eventos en un canal al que se pueden suscribir otros microservicios. El sistema puede agregar nuevos microservicios sin cambiar el servicio de publicación.

Q

plan de consulta

Serie de pasos, como instrucciones, que se utilizan para acceder a los datos de un sistema de base de datos relacional SQL.

regresión del plan de consulta

El optimizador de servicios de la base de datos elige un plan menos óptimo que antes de un cambio determinado en el entorno de la base de datos. Los cambios en estadísticas,

restricciones, configuración del entorno, enlaces de parámetros de consultas y actualizaciones del motor de base de datos PostgreSQL pueden provocar una regresión del plan.

R

Matriz RACI

Consulte responsable, fiable, consultada e informada (RACI).

RAG

Consulte generación aumentada por recuperación.

ransomware

Software malicioso que se ha diseñado para bloquear el acceso a un sistema informático o a los datos hasta que se efectúe un pago.

Matriz RASCI

Consulte responsable, fiable, consultada e informada (RACI).

RCAC

Consulte control de acceso por filas y columnas.

réplica de lectura

Una copia de una base de datos que se utiliza con fines de solo lectura. Puede enrutar las consultas a la réplica de lectura para reducir la carga en la base de datos principal.

rediseñar

Consulte Las 7 R.

objetivo de punto de recuperación (RPO)

La cantidad de tiempo máximo aceptable desde el último punto de recuperación de datos. Esto determina qué se considera una pérdida de datos aceptable entre el último punto de recuperación y la interrupción del servicio.

objetivo de tiempo de recuperación (RTO)

La demora máxima aceptable entre la interrupción del servicio y el restablecimiento del servicio.

refactorizar

Consulte Las 7 R.

Región

Conjunto de recursos de AWS que se encuentran en un área geográfica. Cada Región de AWS está aislada y es independiente de las demás para ofrecer tolerancia a errores, estabilidad y resistencia. Para más información, consulte Specify which Regiones de AWS your account can use.

regresión

Una técnica de ML que predice un valor numérico. Por ejemplo, para resolver el problema de "¿A qué precio se venderá esta casa?", un modelo de ML podría utilizar un modelo de regresión lineal para predecir el precio de venta de una vivienda en función de datos conocidos sobre ella (por ejemplo, los metros cuadrados).

volver a alojar

Consulte Las 7 R.

versión

En un proceso de implementación, el acto de promover cambios en un entorno de producción.

reubicar

Consulte Las 7 R.

redefinir la plataforma

Consulte Las 7 R.

recomprar

Consulte Las 7 R.

resiliencia

Capacidad de una aplicación para resistir interrupciones o recuperarse de ellas. Al planificar la resiliencia en la Nube de AWS, la alta disponibilidad y la recuperación ante desastres son consideraciones comunes. Para más información, consulte Resiliencia en la Nube de AWS.

política basada en recursos

Una política asociada a un recurso, como un bucket de Amazon S3, un punto de conexión o una clave de cifrado. Este tipo de política especifica a qué entidades principales se les permite el acceso, las acciones compatibles y cualquier otra condición que deba cumplirse.

matriz responsable, confiable, consultada e informada (RACI)

Una matriz que define las funciones y responsabilidades de todas las partes involucradas en las actividades de migración y las operaciones de la nube. El nombre de la matriz se deriva de los tipos de responsabilidad definidos en la matriz: responsable (R), contable (A), consultado (C) e informado (I). El tipo de soporte (S) es opcional. Si incluye el soporte, la matriz se denomina matriz RASCI y, si la excluye, se denomina matriz RACI.

control receptivo

Un control de seguridad que se ha diseñado para corregir los eventos adversos o las desviaciones con respecto a su base de seguridad. Para obtener más información, consulte Controles receptivos en Implementación de controles de seguridad en AWS.

retain

Consulte Las 7 R.

retirar

Consulte Las 7 R.

Generación aumentada de recuperación (RAG)

Tecnología de IA generativa mediante la que un LLM hace referencia a un origen de datos autorizado que se encuentra fuera de sus orígenes de datos de entrenamiento antes de generar una respuesta. Por ejemplo, un modelo de RAG podría hacer una búsqueda semántica en la base de conocimientos o en los datos personalizados de una organización. Para más información, consulte ¿Qué es RAG (generación aumentada por recuperación)?

rotación

Proceso mediante el que periódicamente se actualiza un secreto para que resulte más difícil que un atacante pueda acceder a las credenciales.

control de acceso por filas y columnas (RCAC)

El uso de expresiones SQL básicas y flexibles que tienen reglas de acceso definidas. El RCAC consta de permisos de fila y máscaras de columnas.

RPO

Consulte objetivo de punto de recuperación.

RTO

Consulte objetivo de tiempo de recuperación.

manual de procedimientos

Conjunto de procedimientos manuales o automatizados necesarios para realizar una tarea específica. Por lo general, se diseñan para agilizar las operaciones o los procedimientos repetitivos con altas tasas de error.

S

SAML 2.0

Un estándar abierto que utilizan muchos proveedores de identidades (IdP). Esta característica permite el inicio de sesión único (SSO) federado a fin de que los usuarios puedan iniciar sesión en la Consola de administración de AWS o llamar a la API de AWS sin necesidad de crear un usuario de IAM para cada persona de la organización. Para obtener más información sobre la federación basada en SAML 2.0, consulte Acerca de la federación basada en SAML 2.0 en la documentación de IAM.

SCADA

Consulte control de supervisión y adquisición de datos.

SCP

Consulte política de control de servicio.

secreta

En AWS Secrets Manager, información confidencial o restringida, como una contraseña o credenciales de usuario, que almacena de forma cifrada. Se compone del valor del secreto y de sus metadatos. El valor del secreto puede ser binario, una sola cadena o varias cadenas. Para más información, consulte What's in a Secrets Manager secret? en la documentación de Secrets Manager.

seguridad desde el diseño

Enfoque de ingeniería de sistemas que tiene en cuenta la seguridad durante todo el proceso de desarrollo.

control de seguridad

Barrera de protección técnica o administrativa que impide, detecta o reduce la capacidad de un agente de amenazas para aprovechar una vulnerabilidad de seguridad. Existen cuatro tipos de controles de seguridad principales: preventivos, de detección, de respuesta y proactivos.

refuerzo de la seguridad

Proceso de reducir la superficie expuesta a ataques para hacerla más resistente a los ataques. Esto puede incluir acciones, como la eliminación de los recursos que ya no se necesitan, la implementación de prácticas recomendadas de seguridad consistente en conceder privilegios mínimos o la desactivación de características innecesarias en los archivos de configuración.

sistema de información sobre seguridad y administración de eventos (SIEM)

Herramientas y servicios que combinan sistemas de administración de información sobre seguridad (SIM) y de administración de eventos de seguridad (SEM). Un sistema de SIEM recopila, monitorea y analiza los datos de servidores, redes, dispositivos y otras fuentes para detectar amenazas y brechas de seguridad y generar alertas.

automatización de la respuesta de seguridad

Acción predefinida y programada que está diseñada para responder automáticamente a un evento de seguridad o corregirlo. Estas automatizaciones sirven como controles de seguridad de detección o de respuesta que lo ayudan a implementar las prácticas recomendadas de seguridad de AWS. La modificación de un grupo de seguridad de VPC, la aplicación de revisiones a una instancia de Amazon EC2 o la rotación de credenciales son algunos ejemplos de acciones de respuesta automatizadas.

cifrado del servidor

Cifrado de los datos en su destino, por parte del Servicio de AWS que los recibe.

política de control de servicio (SCP)

Una política que proporciona un control centralizado de los permisos de todas las cuentas de una organización en AWS Organizations. Las SCP definen barreras de protección o establecen límites a las acciones que un administrador puede delegar en los usuarios o roles. Puede utilizar las SCP como listas de permitidos o rechazados, para especificar qué servicios o acciones se encuentra permitidos o prohibidos. Para obtener más información, consulte Políticas de control de servicio en la documentación de AWS Organizations.

punto de enlace de servicio

La URL del punto de entrada para un Servicio de AWS. Para conectarse mediante programación a un servicio de destino, puede utilizar un punto de conexión. Para obtener más información, consulte Puntos de conexión de Servicio de AWS en Referencia general de AWS.

acuerdo de nivel de servicio (SLA)

Acuerdo que aclara lo que un equipo de TI se compromete a ofrecer a los clientes, como el tiempo de actividad y el rendimiento del servicio.

indicador de nivel de servicio (SLI)

Medición de un aspecto del rendimiento de un servicio, como la tasa de errores, la disponibilidad o el rendimiento.

objetivo de nivel de servicio (SLO)

Métrica objetivo que representa el estado de un servicio medido mediante un indicador de nivel de servicio.

modelo de responsabilidad compartida

Modelo que describe la responsabilidad que comparte con AWS en cuanto a la conformidad y la seguridad en la nube. AWS es responsable de la seguridad de la nube, mientras que usted es responsable de la seguridad en la nube. Para obtener más información, consulte el Modelo de responsabilidad compartida.

SIEM

Consulte sistema de administración de eventos e información de seguridad.

único punto de error (SPOF)

Error en un único componente crítico de una aplicación que puede interrumpir el sistema.

SLA

Consulte acuerdo de nivel de servicio.

SLI

Consulte indicador de nivel de servicio.

SLO

Consulte objetivo de nivel de servicio.

modelo de dividir y sembrar

Un patrón para escalar y acelerar los proyectos de modernización. A medida que se definen las nuevas funciones y los lanzamientos de los productos, el equipo principal se divide para

crear nuevos equipos de productos. Esto ayuda a ampliar las capacidades y los servicios de su organización, mejora la productividad de los desarrolladores y apoya la innovación rápida. Para más información, consulte Phased approach to modernizing applications in the Nube de AWS.

SPOF

Consulte único punto de error.

esquema en estrella

Estructura organizativa de una base de datos que utiliza una tabla de hechos de gran tamaño para almacenar datos transaccionales o medidos y una o varias tablas dimensionales más pequeñas para almacenar los atributos de los datos. Esta estructura está diseñada para utilizarse en un almacén de datos o con fines de inteligencia empresarial.

patrón de higo estrangulador

Un enfoque para modernizar los sistemas monolíticos mediante la reescritura y el reemplazo gradual de las funciones del sistema hasta que se pueda desmantelar el sistema heredado. Este patrón utiliza la analogía de una higuera que crece hasta convertirse en un árbol estable y, finalmente, se apodera y reemplaza a su host. El patrón fue presentado por Martin Fowler como una forma de gestionar el riesgo al reescribir sistemas monolíticos. Para ver un ejemplo con la aplicación de este patrón, consulte Modernización gradual de los servicios web antiguos de Microsoft ASP.NET (ASMX) mediante contenedores y Amazon API Gateway.

subred

Un intervalo de direcciones IP en la VPC. Una subred debe residir en una sola zona de disponibilidad.

control de supervisión y adquisición de datos (SCADA)

En el sector de la fabricación, sistema que utiliza hardware y software para supervisar los activos físicos y las operaciones de producción.

cifrado simétrico

Un algoritmo de cifrado que utiliza la misma clave para cifrar y descifrar los datos.

pruebas sintéticas

Prueba de un sistema de manera que simule las interacciones de los usuarios para detectar posibles problemas o supervisar el rendimiento. Puede crear estas pruebas mediante Amazon CloudWatch Synthetics.

petición del sistema

Técnica para proporcionar contexto, instrucciones o pautas a un LLM para dirigir su comportamiento. Las peticiones del sistema ayudan a establecer el contexto y las reglas para las interacciones con los usuarios.

T

etiquetas

Pares de clave y valor que funcionan como metadatos para organizar los recursos de AWS. Las etiquetas pueden ayudarle a administrar, identificar, organizar, buscar y filtrar recursos. Para obtener más información, consulte Etiquetado de los recursos de AWS.

variable de destino

El valor que intenta predecir en el ML supervisado. Esto también se conoce como variable de resultado. Por ejemplo, en un entorno de fabricación, la variable objetivo podría ser un defecto del producto.

lista de tareas

Herramienta que se utiliza para hacer un seguimiento del progreso mediante un manual de procedimientos. La lista de tareas contiene una descripción general del manual de procedimientos y una lista de las tareas generales que deben completarse. Para cada tarea general, se incluye la cantidad estimada de tiempo necesario, el propietario y el progreso.

entorno de prueba

Consulte entorno.

entrenamiento

Proporcionar datos de los que pueda aprender su modelo de ML. Los datos de entrenamiento deben contener la respuesta correcta. El algoritmo de aprendizaje encuentra patrones en los datos de entrenamiento que asignan los atributos de los datos de entrada al destino (la respuesta que desea predecir). Genera un modelo de ML que captura estos patrones. Luego, el modelo de ML se puede utilizar para obtener predicciones sobre datos nuevos para los que no se conoce el destino.

puerta de enlace de tránsito

Centro de tránsito de red que puede utilizar para interconectar las VPC y las redes en las instalaciones. Para obtener más información, consulte ¿Qué es una puerta de enlace de tránsito? en la documentación de AWS Transit Gateway.

flujo de trabajo basado en enlaces troncales

Un enfoque en el que los desarrolladores crean y prueban características de forma local en una rama de característica y, a continuación, combinan esos cambios en la rama principal. Luego, la rama principal se adapta a los entornos de desarrollo, preproducción y producción, de forma secuencial.

acceso de confianza

Concesión de permisos a un servicio que especifique para realizar tareas en su empresa en AWS Organizations y en sus cuentas en su nombre. El servicio de confianza crea un rol vinculado al servicio en cada cuenta, cuando ese rol es necesario, para realizar las tareas de administración por usted. Para obtener más información, consulte Uso de AWS Organizations con otros servicios de AWS en la documentación de AWS Organizations.

ajuste

Cambiar aspectos de su proceso de formación a fin de mejorar la precisión del modelo de ML. Por ejemplo, puede entrenar el modelo de ML al generar un conjunto de etiquetas, incorporar etiquetas y, luego, repetir estos pasos varias veces con diferentes ajustes para optimizar el modelo.

equipo de dos pizzas

Un pequeño equipo de DevOps al que puedes alimentar con dos pizzas. Un equipo formado por dos integrantes garantiza la mejor oportunidad posible de colaboración en el desarrollo de software.

U

incertidumbre

Un concepto que hace referencia a información imprecisa, incompleta o desconocida que puede socavar la fiabilidad de los modelos predictivos de ML. Hay dos tipos de incertidumbre: la incertidumbre epistémica se debe a datos limitados e incompletos, mientras que la incertidumbre

aleatoria se debe al ruido y la aleatoriedad inherentes a los datos. Para más información, consulte la guía Cuantificación de la incertidumbre en los sistemas de aprendizaje profundo.

tareas indiferenciadas

También conocido como tareas arduas, es el trabajo que es necesario para crear y operar una aplicación, pero que no proporciona un valor directo al usuario final ni proporciona una ventaja competitiva. Algunos ejemplos de tareas indiferenciadas son la adquisición, el mantenimiento y la planificación de la capacidad.

entornos superiores

Consulte entorno.

V

succión

Una operación de mantenimiento de bases de datos que implica limpiar después de las actualizaciones incrementales para recuperar espacio de almacenamiento y mejorar el rendimiento.

control de versión

Procesos y herramientas que realizan un seguimiento de los cambios, como los cambios en el código fuente de un repositorio.

Interconexión con VPC

Conexión entre dos VPC que permite enrutar el tráfico mediante direcciones IP privadas. Para obtener más información, consulte ¿Qué es una interconexión de VPC? en la documentación de Amazon VPC.

vulnerabilidad

Defecto de software o hardware que pone en peligro la seguridad del sistema.

W

caché caliente

Un búfer caché que contiene datos actuales y relevantes a los que se accede con frecuencia. La instancia de base de datos puede leer desde la caché del búfer, lo que es más rápido que leer desde la memoria principal o el disco.

datos templados

Datos a los que el acceso es infrecuente. Al consultar este tipo de datos, normalmente se aceptan consultas moderadamente lentas.

función de ventana

Función SQL que hace un cálculo en un grupo de filas que se relacionan de alguna manera con el registro actual. Las funciones de ventana son útiles para las tareas de procesamiento, como calcular una media móvil o acceder al valor de las filas en función de la posición relativa de la fila actual.

carga de trabajo

Conjunto de recursos y código que ofrece valor comercial, como una aplicación orientada al cliente o un proceso de backend.

flujo de trabajo

Grupos funcionales de un proyecto de migración que son responsables de un conjunto específico de tareas. Cada flujo de trabajo es independiente, pero respalda a los demás flujos de trabajo del proyecto. Por ejemplo, el flujo de trabajo de la cartera es responsable de priorizar las aplicaciones, planificar las oleadas y recopilar los metadatos de migración. El flujo de trabajo de la cartera entrega estos recursos al flujo de trabajo de migración, que luego migra los servidores y las aplicaciones.

WORM

Consulte escritura única y lectura múltiple.

WQF

Consulte AWS Workload Qualification Framework.

escritura única y lectura múltiple (WORM)

Modelo de almacenamiento que escribe los datos una sola vez y evita que se eliminen o modifiquen. Los usuarios autorizados pueden leer los datos tantas veces como sea necesario,

pero no los pueden cambiar. Esta infraestructura de almacenamiento de datos se considera inmutable.

Z

ataque de día cero

Ataque, normalmente de malware, que se aprovecha de una vulnerabilidad de día cero.

vulnerabilidad de día cero

Un defecto o una vulnerabilidad sin mitigación en un sistema de producción. Los agentes de amenazas pueden usar este tipo de vulnerabilidad para atacar el sistema. Los desarrolladores suelen darse cuenta de la vulnerabilidad a raíz del ataque.

peticiones desde cero

Proporcionar a un LLM instrucciones para llevar a cabo una tarea, pero sin ejemplos (pasos) que puedan ayudar a guiarlo. El LLM debe usar los conocimientos del entrenamiento previo para llevar a cabo la tarea. La eficacia de la petición desde cero depende de la complejidad de la tarea y de la calidad de la petición. Consulte también peticiones con pocos pasos.

aplicación zombi

Aplicación que utiliza un promedio de CPU y memoria menor al 5 por ciento. En un proyecto de migración, es habitual retirar estas aplicaciones.