Cloud-Designmuster, Architekturen und Implementierungen

AWS Präskriptive Leitlinien

AWS Präskriptive Leitlinien: Cloud-Designmuster, Architekturen und Implementierungen

Table of Contents

Cloud-Designmuster, Architekturen und Implementierungen

Anitha Deenadayalan, Amazon Web Services (AWS)

Mai 2024 (Dokumentenhistorie)

Dieser Leitfaden enthält Anleitungen zur Implementierung häufig verwendeter Entwurfsmuster für Modernisierungen mithilfe von AWS Diensten. Immer mehr moderne Anwendungen werden unter Verwendung von Microservices-Architekturen entwickelt, um Skalierbarkeit zu erreichen, die Release-Geschwindigkeit zu verbessern, die Auswirkungen von Änderungen zu verringern und Regression zu reduzieren. Dies führt zu einer verbesserten Entwicklerproduktivität und erhöhter Agilität, besserer Innovation und einer stärkeren Konzentration auf die Geschäftsanforderungen. Microservices-Architekturen unterstützen auch die Verwendung der optimalen Technologie für den Service und die Datenbank und fördern polyglotten Code und polyglotte Persistenz.

Traditionell laufen monolithische Anwendungen in einem einzigen Prozess, verwenden einen Datenspeicher und laufen auf Servern, die vertikal skaliert werden. Im Vergleich dazu sind moderne Microservice-Anwendungen differenziert, haben unabhängige Fehlerdomains, laufen als Services über das Netzwerk und können je nach Anwendungsfall mehr als einen Datenspeicher verwenden. Die Services sind horizontal skalierbar, und eine einzige Transaktion kann sich über mehrere Datenbanken erstrecken. Entwicklungsteams müssen sich bei der Entwicklung von Anwendungen mithilfe von Microservices-Architekturen auf Netzwerkkommunikation, polyglotte Persistenz, horizontale Skalierung, Ereigniskonsistenz und Transaktionsverarbeitung über die Datenspeicher hinweg konzentrieren. Daher sind Modernisierungsmuster für die Lösung häufig auftretender Probleme in der modernen Anwendungsentwicklung von entscheidender Bedeutung und tragen dazu bei, die Bereitstellung von Software zu beschleunigen.

Dieser Leitfaden bietet eine technische Referenz für Cloud-Architekten, technische Führungskräfte, Anwendungs- und Geschäftsverantwortliche sowie Entwickler, die die richtige Cloud-Architektur für Designmuster auf der Grundlage bewährter Methoden auswählen möchten. Jedes in diesem Leitfaden besprochene Muster befasst sich mit einem oder mehreren bekannten Szenarien in Microservices-Architekturen. Der Leitfaden erörtert die mit den einzelnen Mustern verbundenen Probleme und Überlegungen, bietet eine hochrangige Architekturimplementierung und beschreibt die AWS-Implementierung für das Muster. GitHub Open-Source-Beispiele und Workshop-Links werden, sofern verfügbar, bereitgestellt.

Der Leitfaden behandelt die folgenden Muster:

- Anti-Korruptions-Ebene
- API-Routing-Muster:
  - Hostnamen-Routing
  - Pfad-Routing
  - HTTP-Header-Routing
- Leistungsschutzschalter
- Ereignis-Sourcing
- Sechseckige Architektur
- Publish-Subscribe
- Wiederholungsversuch mit Backoff
- Saga-Muster:
  - Saga-Choreographie
  - Saga-Orchestrierung
- Verstreuen und sammeln
- Strangler Fig
- Transactional Outbox

Gezielte Geschäftsergebnisse

Wenn Sie die in diesem Leitfaden beschriebenen Muster zur Modernisierung Ihrer Anwendungen verwenden, können Sie:

- Zuverlässige, sichere und betrieblich effiziente Architekturen entwerfen und implementieren, die hinsichtlich Kosten und Leistung optimiert sind.
- Die Zykluszeit für Anwendungsfälle, die diese Muster erfordern, verkürzen, so dass Sie sich stattdessen auf organisationsspezifische Herausforderungen konzentrieren können.
- Die Entwicklung beschleunigen, indem Sie die Implementierung von Mustern mithilfe von AWS-Services standardisieren.
- Ihre Entwickler dabei unterstützen, moderne Anwendungen zu entwickeln, ohne technische Schulden zu vererben.

Ebenenmuster zur Korruptionsbekämpfung

Absicht

Das Anti-Korruptions-Layer-Muster (ACL) fungiert als Vermittlungsschicht, die die Semantik des Domänenmodells von einem System in ein anderes System übersetzt. Es übersetzt das Modell des Upstream-begrenzten Kontextes (Monolith) in ein Modell, das für den Downstream-begrenzten Kontext (Microservice) geeignet ist, bevor der vom Upstream-Team erstellte Kommunikationsvertrag genutzt wird. Dieses Muster kann anwendbar sein, wenn der begrenzte Downstream-Kontext eine Kern-Subdomäne enthält oder das Upstream-Modell ein unveränderbares Altsystem ist. Außerdem werden Transformationsrisiken und Betriebsunterbrechungen reduziert, indem verhindert wird, dass Anrufer Änderungen vornehmen, wenn ihre Anrufe transparent an das Zielsystem weitergeleitet werden müssen.

Motivation

Während des Migrationsprozesses, wenn eine monolithische Anwendung in Microservices migriert wird, kann es zu Änderungen in der Semantik des Domänenmodells des neu migrierten Dienstes kommen. Wenn die Funktionen innerhalb des Monolithen erforderlich sind, um diese Microservices aufzurufen, sollten die Anrufe an den migrierten Dienst weitergeleitet werden, ohne dass Änderungen an den Anrufdiensten erforderlich sind. Das ACL-Muster ermöglicht es dem Monolith, die Microservices transparent aufzurufen, indem er als Adapter oder Fassadenschicht fungiert, die die Aufrufe in die neuere Semantik übersetzt.

Anwendbarkeit

Erwägen Sie, dieses Muster zu verwenden, wenn:

- Ihre bestehende monolithische Anwendung muss mit einer Funktion kommunizieren, die in einen Microservice migriert wurde, und das migrierte Dienstdomänenmodell und die Semantik unterscheiden sich vom ursprünglichen Feature.
- Zwei Systeme haben unterschiedliche Semantik und müssen Daten austauschen, aber es ist nicht praktikabel, ein System so zu modifizieren, dass es mit dem anderen System kompatibel ist.
- Sie möchten einen schnellen und vereinfachten Ansatz verwenden, um ein System mit minimalen Auswirkungen an ein anderes anzupassen.

- Ihre Anwendung kommuniziert mit einem externen System.

Probleme und Überlegungen

- Teamabhängigkeiten:Wenn verschiedene Dienste in einem System verschiedenen Teams gehören, kann die Semantik des neuen Domänenmodells in den migrierten Diensten zu Änderungen in den aufrufenden Systemen führen. Teams sind jedoch möglicherweise nicht in der Lage, diese Änderungen auf koordinierte Weise vorzunehmen, da sie möglicherweise andere Prioritäten haben. ACL entkoppelt die Anrufer und übersetzt die Aufrufe so, dass sie der Semantik der neuen Dienste entsprechen, sodass die Anrufer keine Änderungen am aktuellen System vornehmen müssen.
- Operativer Aufwand:Das ACL-Muster erfordert zusätzlichen Aufwand für Betrieb und Wartung. Diese Arbeit umfasst die Integration von ACL in Überwachungs- und Warnungstools, den Release-Prozess sowie Prozesse für kontinuierliche Integration und kontinuierliche Bereitstellung (CI/CD).
- Ein einziger Fehlerpunkt:Fehler in der ACL können dazu führen, dass der Zieldienst nicht erreichbar ist, was zu Anwendungsproblemen führt. Um dieses Problem zu beheben, sollten Sie Wiederholungsfunktionen und Schutzschalter einbauen. Sehen Sie diemit Backoff erneut versuchenundSchutzschalterMuster, um mehr über diese Optionen zu erfahren. Durch die Einrichtung geeigneter Warnmeldungen und Protokollierung wird die durchschnittliche Lösungszeit (MTTR) verbessert.
- Technische Schulden:Überlegen Sie im Rahmen Ihrer Migrations- oder Modernisierungsstrategie, ob es sich bei der ACL um eine vorübergehende oder vorläufige Lösung oder um eine langfristige Lösung handelt. Wenn es sich um eine Zwischenlösung handelt, sollten Sie die ACL als technische Schuld erfassen und sie außer Betrieb nehmen, nachdem alle abhängigen Anrufer migriert wurden.
- Latenz:Die zusätzliche Schicht kann aufgrund der Konvertierung von Anfragen von einer Schnittstelle zur anderen zu Latenz führen. Wir empfehlen, dass Sie die Leistungstoleranz in Anwendungen definieren und testen, die empfindlich auf die Reaktionszeit reagieren, bevor Sie ACL in Produktionsumgebungen einsetzen.
- Skalierungsengpass:In Anwendungen mit hoher Auslastung, bei denen Dienste bis zur Spitzenlast skaliert werden können, kann ACL zu einem Engpass werden und zu Skalierungsproblemen führen. Wenn der Zieldienst nach Bedarf skaliert wird, sollten Sie ACL so gestalten, dass es entsprechend skaliert.
- Servicespezifische oder gemeinsame Implementierung:Sie können ACL als gemeinsames Objekt entwerfen, um Aufrufe zu konvertieren und an mehrere Dienste oder dienstspezifische

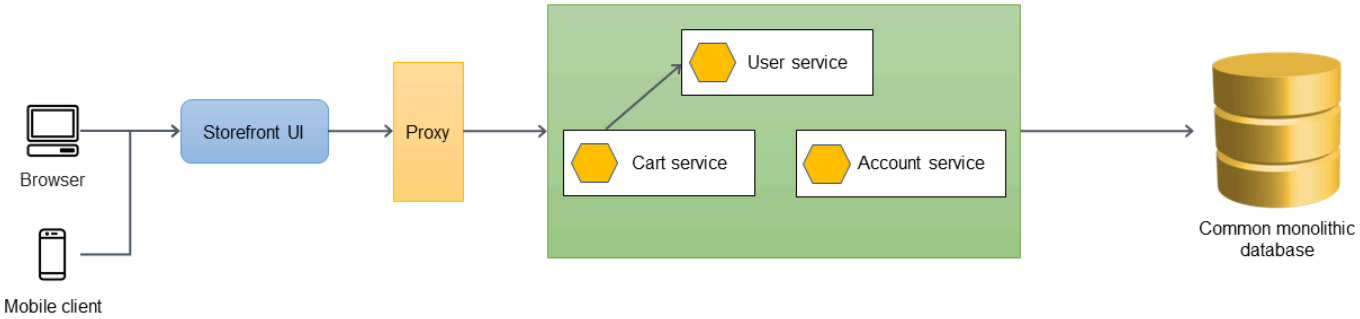
Klassen weiterzuleiten. Berücksichtigen Sie Latenz, Skalierung und Fehlertoleranz, wenn Sie den Implementierungstyp für ACL festlegen.

Implementierung

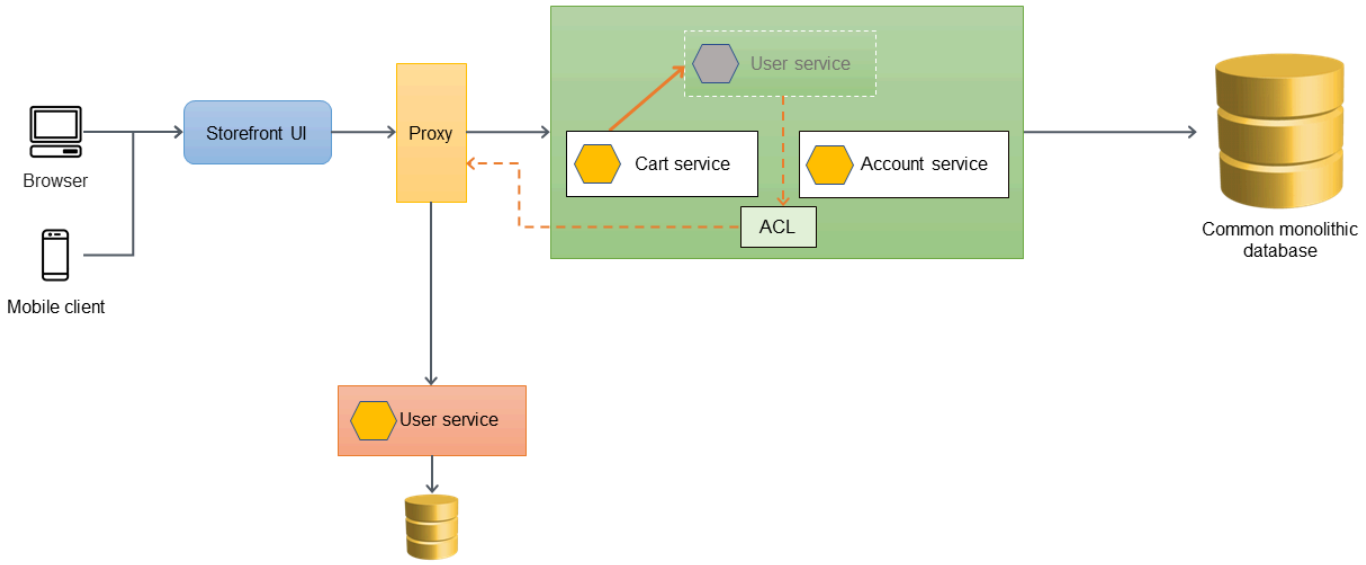
Sie können ACL in Ihrer monolithischen Anwendung als Klasse implementieren, die spezifisch für den Dienst ist, der migriert wird, oder als unabhängigen Dienst. Die ACL muss außer Betrieb genommen werden, nachdem alle abhängigen Dienste in die Microservices-Architektur migriert wurden.

Hochrangige Architektur

In der folgenden Beispielarchitektur verfügt eine monolithische Anwendung über drei Dienste: Benutzerservice, Warenkorbservice und Kontoservice. Der Warenkorb-Service ist vom Benutzerdienst abhängig, und die Anwendung verwendet eine monolithische relationale Datenbank.

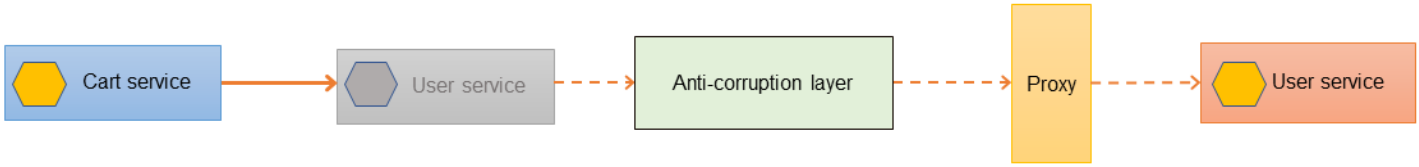


In der folgenden Architektur wurde der Benutzerdienst auf einen neuen Microservice migriert. Der Warenkorb-Service ruft den Benutzerservice auf, aber die Implementierung ist innerhalb des Monolithen nicht mehr verfügbar. Es ist auch wahrscheinlich, dass die Schnittstelle des neu migrierten Dienstes nicht mit der vorherigen Schnittstelle übereinstimmt, als sie sich in der monolithischen Anwendung befand.



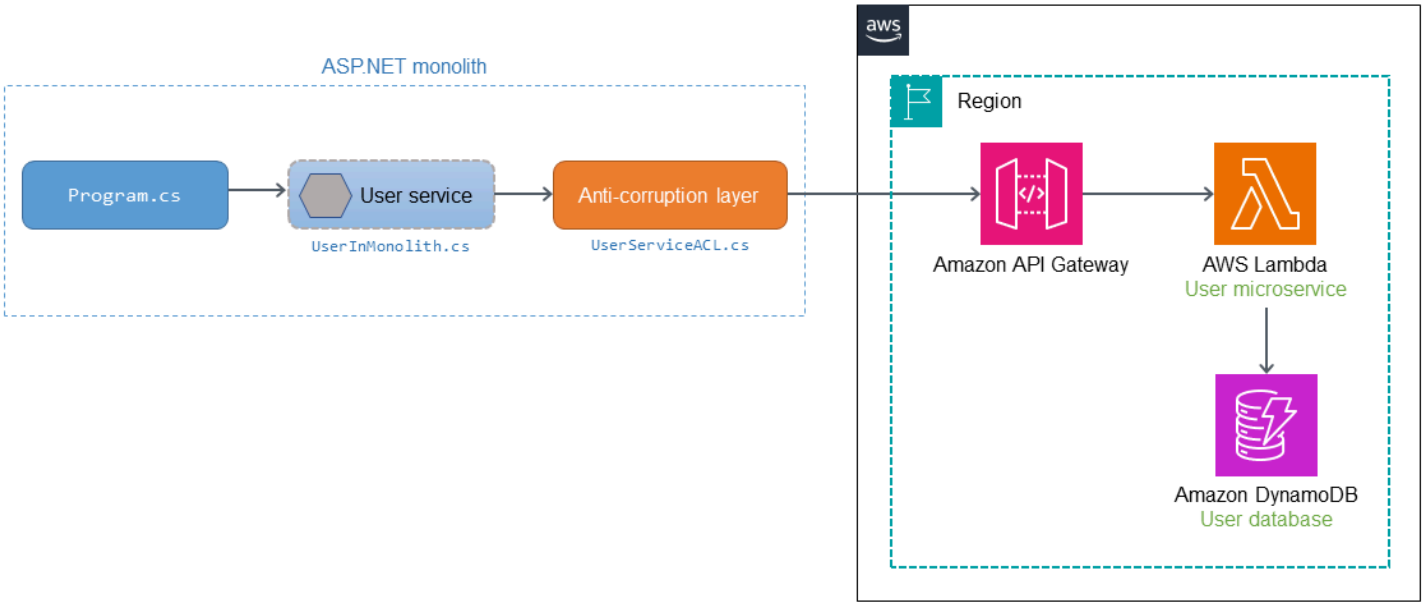
Wenn der Warenkorb-Service den neu migrierten Benutzerservice direkt aufrufen muss, sind Änderungen am Warenkorb-Service und ein gründliches Testen der monolithischen Anwendung erforderlich. Dies kann das Transformationsrisiko und die Geschäftsunterbrechung erhöhen. Ziel sollte es sein, die Änderungen an der bestehenden Funktionalität der monolithischen Anwendung zu minimieren.

In diesem Fall empfehlen wir, eine ACL zwischen dem alten Benutzerdienst und dem neu migrierten Benutzerdienst einzuführen. Die ACL fungiert als Adapter oder Fassade, die die Aufrufe in die neuere Schnittstelle umwandelt. ACL kann innerhalb der monolithischen Anwendung als Klasse implementiert werden (zum Beispiel`UserServiceFacade`oder`UserServiceAdapter`), das spezifisch für den Dienst ist, der migriert wurde. Die Antikorruptionsebene muss außer Betrieb genommen werden, nachdem alle abhängigen Dienste in die Microservices-Architektur migriert wurden.



Umsetzung mitAWSDienstleistungen

Das folgende Diagramm zeigt, wie Sie dieses ACL-Beispiel implementieren können, indem SieAWSdienstleistungen.



Der Benutzer-Microservice wird aus der monolithischen ASP.NET-Anwendung migriert und alsAWS LambdaFunktion auf AWS. Aufrufe an die Lambda-Funktion werden weitergeleitetAmazon-API-Gateway. ACL wird im Monolith eingesetzt, um den Aufruf zu übersetzen und so an die Semantik des Benutzer-Microservices anzupassen.

Wann`Program.cs`ruft den Benutzerdienst auf (`UserInMonolith.cs`) innerhalb des Monolithen wird der Anruf an die ACL weitergeleitet (`UserServiceACL.cs`). Die ACL übersetzt den Aufruf in die neue Semantik und Schnittstelle und ruft den Microservice über den API Gateway-Endpunkt auf. Der Anrufer (`Program.cs`) ist sich der Übersetzung und des Routings, die im Benutzerservice und ACL stattfinden, nicht bewusst. Da der Anrufer sich der Codeänderungen nicht bewusst ist, kommt es zu weniger Betriebsunterbrechungen und einem geringeren Transformationsrisiko.

Beispiel-Code

Der folgende Codeausschnitt enthält die Änderungen am ursprünglichen Dienst und die Implementierung von`UserServiceACL.cs`. Wenn eine Anfrage eingeht, ruft der ursprüngliche Benutzerdienst die ACL auf. Die ACL konvertiert das Quellobjekt so, dass es der Schnittstelle des neu migrierten Dienstes entspricht, ruft den Dienst auf und gibt die Antwort an den Aufrufer zurück.

```
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;
    }
}
```

GitHubEndlager

Eine vollständige Implementierung der Beispielarchitektur für dieses Muster finden Sie in derGitHubRepositorium beihttps://github.com/aws-samples/anti-corruption-layer-pattern.

Verwandter Inhalt

- Strangler-Feigenmuster
- Schaltschalter-Muster
- Versuchen Sie es erneut mit dem Backoff-Muster

API-Routing-Muster

In agilen Entwicklungsumgebungen besitzen autonome Teams (z. B. Squads und Tribes) einen oder mehrere Services, die viele Microservices umfassen. Die Teams stellen diese Services als APIs zur Verfügung, damit ihre Verbraucher mit ihrer Gruppe von Services und Aktionen interagieren können.

Es gibt drei Hauptmethoden, um HTTP-APIs mit Hilfe von Hostnamen und Pfaden für Upstream-Verbraucher zugänglich zu machen:

Methode	Beschreibung	Beispiel
Hostnamen-Routing	Stellt jeden Service als Hostnamen bereit.	`billing.api.example.com`
Pfad-Routing	Stellt jeden Service als Pfad bereit.	`api.example.com/billing`
Header-basiertes Routing	Stellt jeden Service als HTTP-Header bereit.	`x-example-action: something`

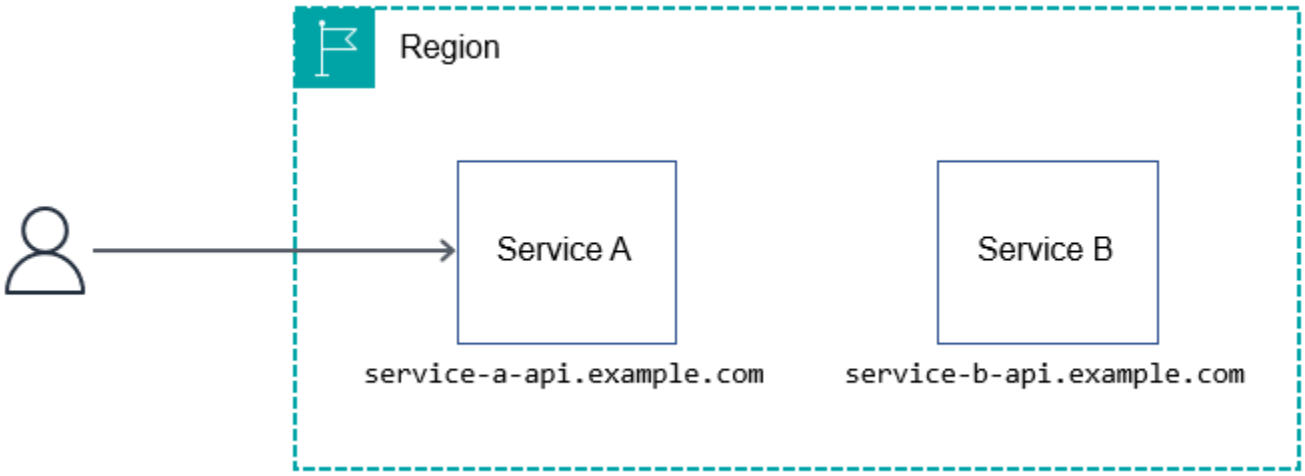
In diesem Abschnitt werden typische Anwendungsfälle für diese drei Routing-Methoden und ihre Vorteile beschrieben, damit Sie entscheiden können, welche Methode am besten zu Ihren Anforderungen und Ihrer Organisationsstruktur passt.

Hostname-Routing-Muster

Das Routing nach Hostnamen ist ein Mechanismus zur Isolierung von API-Services, indem jeder API ein eigener Hostname zugewiesen wird, z. B. `service-a.api.example.com` oder `service-a.example.com`.

Typische Anwendungsfälle

Das Routing unter Verwendung von Hostnamen reduziert die Reibungsverluste in Versionen, da nichts zwischen den Serviceteams ausgetauscht wird. Die Teams sind dafür verantwortlich, alles zu verwalten, von DNS-Einträgen bis hin zu Servicevorgängen in der Produktion.



Vorteile

Hostname-Routing ist bei weitem die einfachste und skalierbarste Methode für HTTP-API-Routing. Sie können jeden relevanten AWS Service verwenden, um eine Architektur zu erstellen, die dieser Methode folgt — Sie können eine Architektur mit Amazon API Gateway AWS AppSync, Application Load Balancers und Amazon Elastic Compute Cloud (Amazon EC2) oder einem anderen HTTP-kompatiblen Service erstellen.

Teams können Hostnamen-Routing verwenden, um ihre Subdomain vollständig zu besitzen. Es macht es auch einfacher, Bereitstellungen für bestimmte AWS-Regionen Versionen zu isolieren, zu testen und zu orchestrieren, z. B. für. `region.service-a.api.example.com` `dev.region.service-a.api.example.com`

Nachteile

Wenn Sie Hostnamen-Routing verwenden, müssen sich Ihre Verbraucher unterschiedliche Hostnamen merken, um mit den einzelnen APIs, die Sie bereitstellen, interagieren zu können. Sie können dieses Problem beheben, indem Sie ein Client-SDK bereitstellen. Client-SDKs bringen jedoch ihre eigenen Herausforderungen mit sich. Sie müssen zum Beispiel fortlaufende Updates, mehrere Sprachen, die Versionsverwaltung, die Übermittlung von fehlerhaften Änderungen aufgrund von Sicherheitsproblemen oder Fehlerkorrekturen, die Dokumentation und so weiter unterstützen.

Wenn Sie Hostnamen-Routing verwenden, müssen Sie außerdem die Subdomain oder Domain jedes Mal registrieren, wenn Sie einen neuen Service erstellen.

Pfad-Routing-Muster

Routing nach Pfaden ist der Mechanismus, mehrere oder alle APIs unter demselben Hostnamen zu gruppieren und einen Anforderungs-URI zu verwenden, um Services zu isolieren; z. B. `api.example.com/service-a` oder `api.example.com/service-b`.

Typische Anwendungsfälle

Die meisten Teams entscheiden sich für diese Methode, weil sie eine einfache Architektur wollen – ein Entwickler muss sich nur eine URL merken, zum Beispiel `api.example.com`, um mit der HTTP-API zu interagieren. Die API-Dokumentation ist oft leichter zu verarbeiten, da sie oft zusammen gehalten wird, anstatt auf verschiedene Portale oder PDFs aufgeteilt zu sein.

Pfadbasiertes Routing gilt als einfacher Mechanismus für die gemeinsame Nutzung einer HTTP-API. Es ist jedoch mit betrieblichem Aufwand wie Konfiguration, Autorisierung, Integrationen und zusätzlicher Latenz aufgrund mehrerer Übergaben verbunden. Außerdem sind ausgereifte Änderungsmanagementprozesse erforderlich, um sicherzustellen, dass eine Fehlkonfiguration nicht zu einer Unterbrechung aller Services führt.

Bei AWS dieser Option gibt es mehrere Möglichkeiten, eine API gemeinsam zu nutzen und effektiv zum richtigen Service weiterzuleiten. In den folgenden Abschnitten werden drei Ansätze behandelt: HTTP Service Reverse Proxy, API Gateway und Amazon CloudFront. Keiner der vorgeschlagenen Ansätze zur Vereinheitlichung von API-Services stützt sich auf die Downstream-Services, die auf AWS laufen. Die Services können überall ohne Probleme oder mit jeder Technologie ausgeführt werden, solange sie HTTP-kompatibel sind.

HTTP-Service-Reverse-Proxy

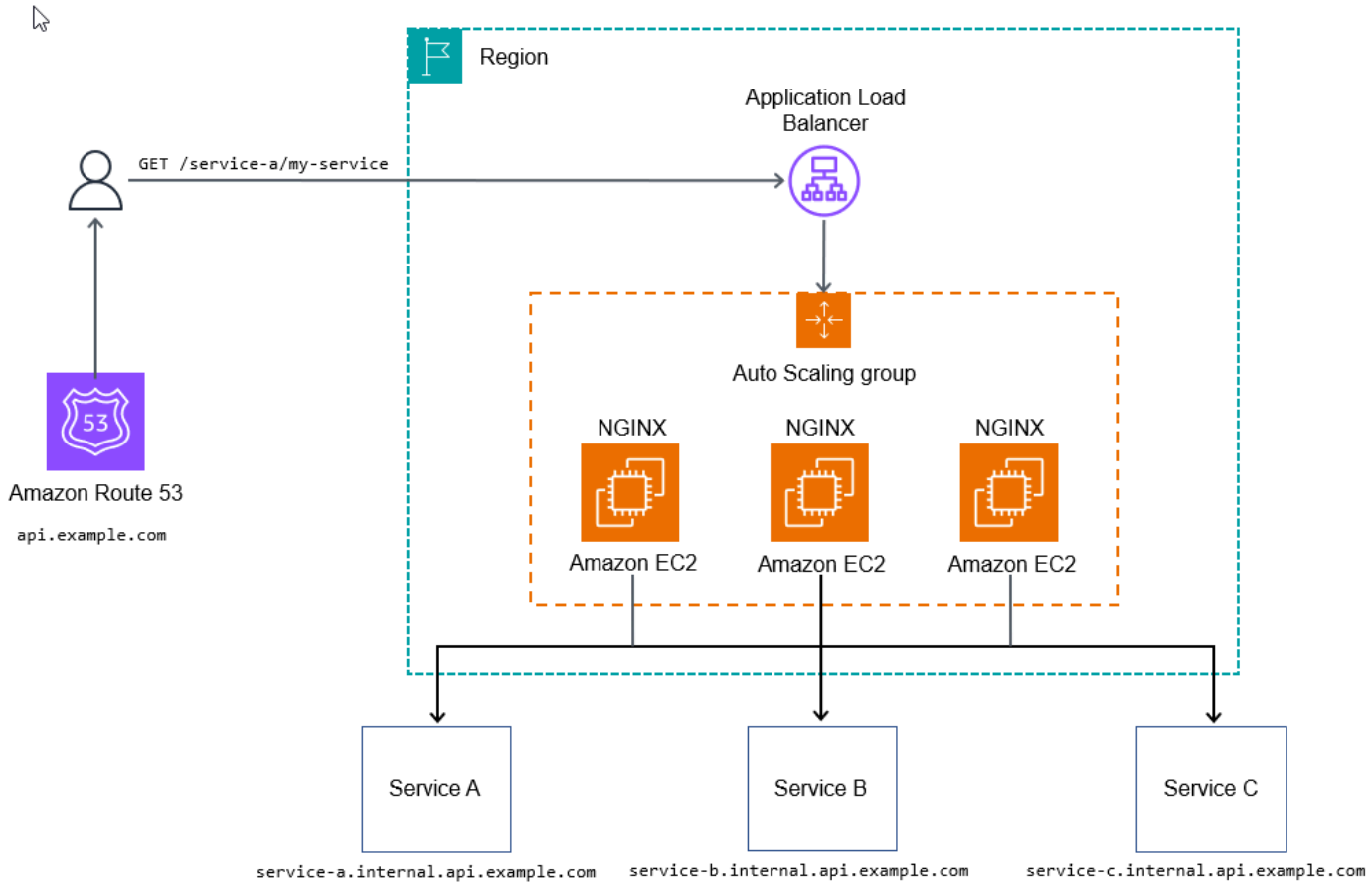
Sie können einen HTTP-Server wie NGINX verwenden, um dynamische Routing-Konfigurationen zu erstellen. In einer Kubernetes-Architektur können Sie auch eine Eingangsregel erstellen, die dem Pfad zu einem Service entspricht. (Dieser Leitfaden behandelt nicht den Kubernetes-Eingang. Weitere Informationen finden Sie in der Kubernetes-Dokumentation.)

Die folgende Konfiguration für NGINX ordnet dynamisch eine HTTP-Anfrage von `api.example.com/my-service/` zu `my-service.internal.api.example.com` zu.

```
server {
    listen  80;
```

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

Das folgende Diagramm veranschaulicht die Methode des HTTP-Service-Reverse-Proxy.



Dieser Ansatz kann für einige Anwendungsfälle ausreichend sein, in denen keine zusätzlichen Konfigurationen verwendet werden, um die Verarbeitung von Anfragen zu starten, sodass die nachgelagerte API Metriken und Protokolle sammeln kann.

Um für die Produktion betriebsbereit zu sein, müssen Sie in der Lage sein, die Beobachtbarkeit auf jeder Ebene Ihres Stacks hinzuzufügen, zusätzliche Konfigurationen vorzunehmen oder Skripte hinzuzufügen, um Ihren API-Eingangspunkt so anzupassen, dass fortgeschrittene Features wie Ratenbegrenzung oder Nutzungs-Tokens möglich sind.

Vorteile

Das ultimative Ziel der Methode des HTTP-Service-Reverse-Proxy ist es, einen skalierbaren und verwaltbaren Ansatz zur Vereinheitlichung von APIs in einer einzigen Domain zu schaffen, so dass sie für jeden API-Verbraucher kohärent erscheint. Dieser Ansatz ermöglicht es Ihren Serviceteams auch, ihre eigenen APIs mit minimalem Aufwand nach der Bereitstellung bereitzustellen und zu verwalten. AWS verwaltete Dienste für die Nachverfolgung, wie AWS X-Rayoder AWS WAF, sind hier weiterhin anwendbar.

Nachteile

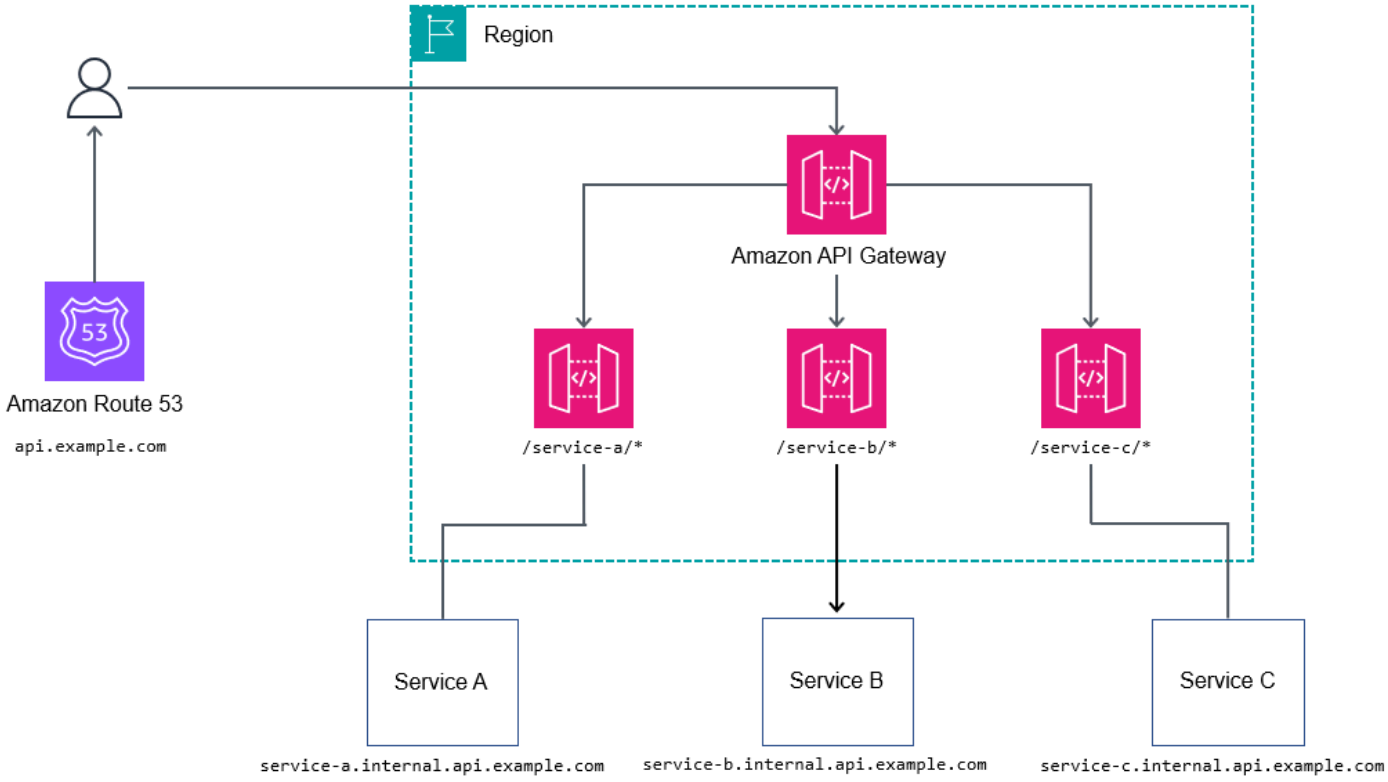
Der größte Nachteil dieses Ansatzes ist das umfangreiche Testen und Verwalten der erforderlichen Infrastrukturkomponenten, obwohl dies möglicherweise kein Problem darstellt, wenn Sie über Teams für Site Reliability Engineering (SRE) verfügen.

Bei dieser Methode gibt es einen Kostenschwellenwert. Bei niedrigen bis mittleren Mengen ist sie teurer als einige der anderen Methoden, die in diesem Handbuch beschrieben werden. Bei hohen Volumen ist sie sehr kostengünstig (etwa 100 000 Transaktionen pro Sekunde oder besser).

API Gateway

Der Amazon-API-Gateway-Service (REST-APIs und HTTP-APIs) kann den Datenverkehr auf eine Weise weiterleiten, die der Methode des HTTP-Service-Reverse-Proxys ähnelt. Die Verwendung eines API-Gateways im HTTP-Proxymodus bietet eine einfache Möglichkeit, viele Services in einen Einstiegspunkt zur Top-Level-Subdomain `api.example.com` einzubinden und dann Anfragen an den verschachtelten Service weiterzuleiten, zum Beispiel `billing.internal.api.example.com`.

Sie möchten wahrscheinlich nicht zu differenziert vorgehen und jeden Pfad in jedem Service im Root- oder Core-API-Gateway abbilden. Entscheiden Sie sich stattdessen für Wildcard-Pfade, wie z. B. `/billing/*`, um Anfragen an den Abrechnungsservice weiterzuleiten. Indem Sie nicht jeden Pfad im Root- oder Core-API-Gateway zuordnen, gewinnen Sie mehr Flexibilität bei Ihren APIs, da Sie das Root-API-Gateway nicht bei jeder API-Änderung aktualisieren müssen.



Vorteile

Für die Kontrolle über komplexere Workflows, wie z. B. die Änderung von Anfrageattributen, stellen REST-APIs die Apache Velocity Template Language (VTL) zur Verfügung, mit der Sie die Anfrage und die Antwort ändern können. REST-APIs können zusätzliche Vorteile wie die folgenden bieten:

- Authentifizierung N/Z mit AWS Identity and Access Management (IAM), Amazon Cognito oder Autorisierern AWS Lambda
- AWS X-Ray zur Rückverfolgung
- Integration mit AWS WAF
- Grundsätzliche Ratenbegrenzung
- Nutzungs-Tokens für die Einteilung der Verbraucher in verschiedene Buckets (siehe Drosselung von API-Anfragen für besseren Durchsatz in der API-Gateway-Dokumentation)

Nachteile

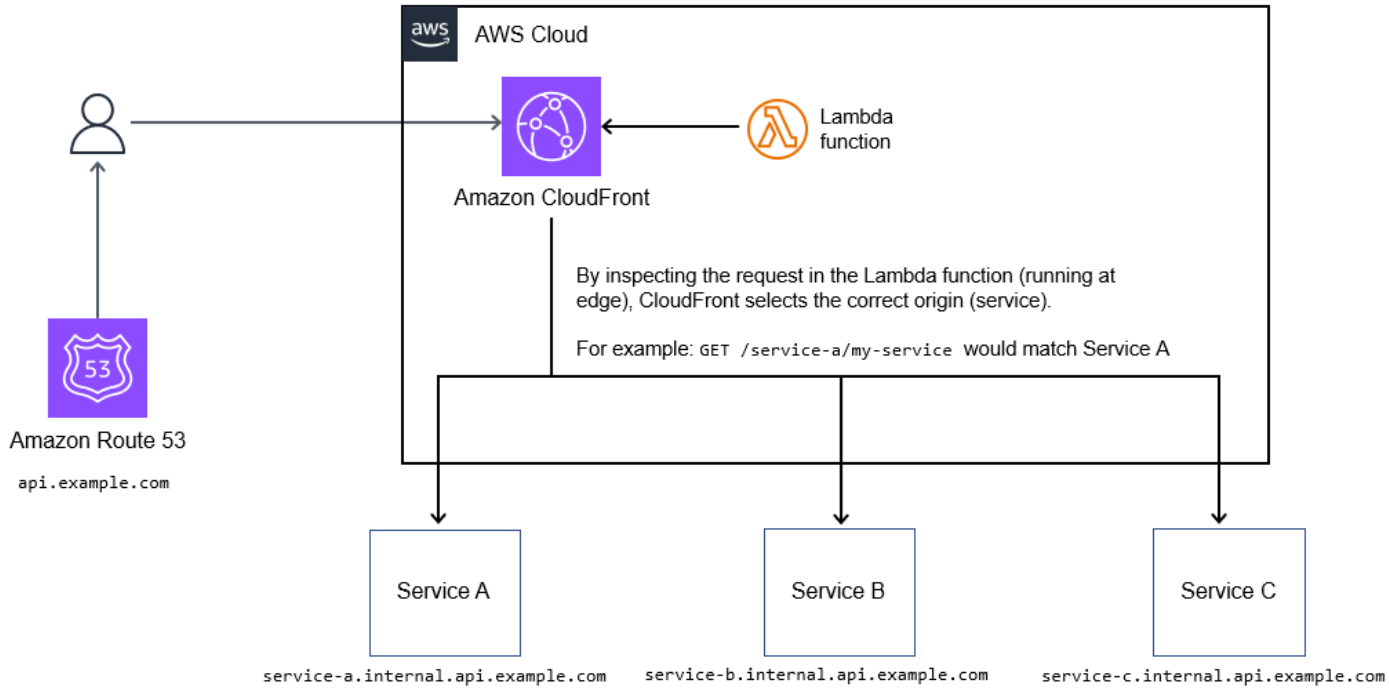
Bei hohem Volumen könnten die Kosten für einige Benutzer ein Problem darstellen.

CloudFront

Sie können die dynamische Herkunftsauswahlfunktion in Amazon verwenden CloudFront, um bedingt einen Ursprung (einen Service) auszuwählen, um die Anfrage weiterzuleiten. Sie können dieses Feature verwenden, um eine Reihe von Services über einen einzigen Hostnamen weiterzuleiten, z. B. `api.example.com`.

Typische Anwendungsfälle

Die Routing-Logik ist als Code in der Lambda@Edge-Funktion enthalten und unterstützt daher hochgradig anpassbare Routing-Mechanismen wie A/B-Tests, Canary-Releases, Feature-Flagging und Pfadumschreibung. Dies wird im folgenden Diagramm veranschaulicht.



Vorteile

Wenn Sie API-Antworten zwischenspeichern müssen, ist diese Methode eine gute Möglichkeit, eine Sammlung von Services hinter einem einzigen Endpunkt zu vereinen. Es ist eine kostengünstige Methode, um Sammlungen von APIs zu vereinheitlichen.

CloudFront Unterstützt außerdem die Verschlüsselung auf Feldebene sowie die Integration mit grundlegenden AWS WAF Ratenbegrenzungen und Basis-ACLs.

Nachteile

Diese Methode unterstützt maximal 250 Ursprünge (Services), die vereinheitlicht werden können. Dieses Limit ist für die meisten Bereitstellungen ausreichend, kann jedoch zu Problemen mit einer großen Anzahl von APIs führen, wenn Sie Ihr Serviceportfolio erweitern.

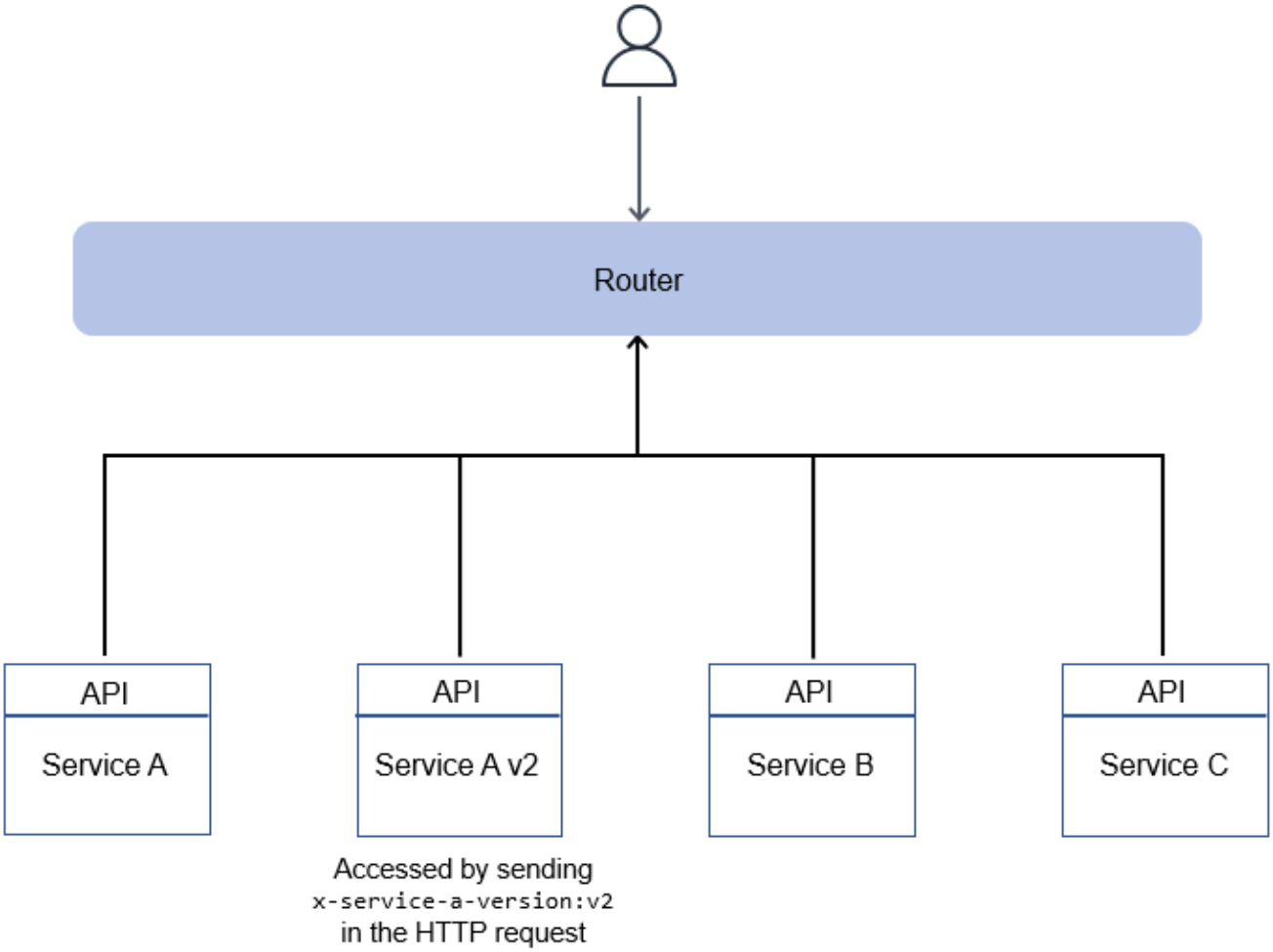
Die Aktualisierung der Lambda @Edge -Funktionen dauert derzeit einige Minuten. CloudFront es dauert außerdem bis zu 30 Minuten, bis die Übertragung der Änderungen an allen Präsenzpunkten abgeschlossen ist. Dadurch werden letztlich weitere Updates blockiert, bis sie abgeschlossen sind.

HTTP-Header-Routing-Muster

Header-basiertes Routing ermöglicht es Ihnen, für jede Anfrage den richtigen Service als Ziel festzulegen, indem Sie in der HTTP-Anfrage einen HTTP-Header angeben. Wenn Sie den Header `x-service-a-action: get-thing` senden, können Sie beispielsweise `get thing` von `Service A` abrufen. Der Pfad der Anfrage ist immer noch wichtig, da er Hinweise darauf bietet, an welcher Ressource Sie arbeiten möchten.

Sie können das HTTP-Header-Routing nicht nur für Aktionen verwenden, sondern auch als Mechanismus für das Versions-Routing, das Aktivieren von Feature-Flagen, A/B-Tests oder ähnlichen Anforderungen. In der Realität werden Sie Header-Routing wahrscheinlich zusammen mit einer der anderen Routing-Methoden verwenden, um robuste APIs zu erstellen.

Die Architektur für HTTP-Header-Routing hat typischerweise eine einfache Routing-Schicht vor den Microservices, die zum richtigen Service weiterleitet und eine Antwort zurückgibt, wie im folgenden Diagramm dargestellt. Diese Routing-Schicht könnte alle oder nur einige Services umfassen, um einen Vorgang wie das versionsbasierte Routing zu ermöglichen.



Vorteile

Konfigurationsänderungen erfordern nur minimalen Aufwand und können einfach automatisiert werden. Außerdem ist diese Methode flexibel und bietet kreative Möglichkeiten, um lediglich bestimmte Vorgänge, die Sie von einem Service erwarten, darzustellen.

Nachteile

Wie bei der Hostnamen-Routing-Methode geht das HTTP-Header-Routing davon aus, dass Sie die volle Kontrolle über den Client haben und benutzerdefinierte HTTP-Header bearbeiten können. Proxys, Content Delivery Networks (CDNs) und Load Balancer können die Header-Größe einschränken. Obwohl dies wahrscheinlich nicht bedenklich ist, könnte es ein Problem darstellen, je nachdem, wie viele Header und Cookies Sie hinzufügen.

Muster des Leistungsschalters

Absicht

Das Circuit Breaker-Muster kann verhindern, dass ein Anruferdienst erneut versucht, einen anderen Dienst (Anrufer) anzurufen, obwohl der Anruf zuvor zu wiederholten Timeouts oder Ausfällen geführt hat. Das Muster wird auch verwendet, um zu erkennen, wann der angerufene Dienst wieder funktionsfähig ist.

Motivation

Wenn mehrere Microservices zusammenarbeiten, um Anfragen zu bearbeiten, kann es sein, dass ein oder mehrere Dienste nicht verfügbar sind oder eine hohe Latenz aufweisen. Wenn komplexe Anwendungen Microservices verwenden, kann ein Ausfall eines Microservices zum Ausfall der Anwendung führen. Microservices kommunizieren über Remote-Prozeduraufrufe, und es können vorübergehende Fehler bei der Netzwerkkonnektivität auftreten, die zu Ausfällen führen. (Die vorübergehenden Fehler können mithilfe des Musters „Wiederholungsversuch mit Backoff“ behandelt werden.) Während der synchronen Ausführung kann die Kaskadierung von Timeouts oder Ausfällen zu einer schlechten Benutzererfahrung führen.

In einigen Situationen kann es jedoch länger dauern, bis die Fehler behoben sind, z. B. wenn der aufgerufene Dienst ausgefallen ist oder ein Datenbankkonflikt zu Timeouts führt. In solchen Fällen, wenn der aufrufende Dienst die Aufrufe wiederholt wiederholt, können diese Wiederholungen zu Netzwerkkonflikten und zur Auslastung des Datenbank-Threadpools führen. Wenn mehrere Benutzer die Anwendung wiederholt wiederholen, verschlimmert sich das Problem zusätzlich und kann zu Leistungseinbußen in der gesamten Anwendung führen.

Das Circuit Breaker Pattern wurde von Michael Nygard in seinem Buch Release It (Nygard 2018) populär gemacht. Dieses Entwurfsmuster kann verhindern, dass ein Anruferdienst einen Serviceanruf wiederholt, der zuvor zu wiederholten Timeouts oder Ausfällen geführt hat. Es kann auch erkennen, wann der angerufene Dienst wieder funktionsfähig ist.

Schutzschalterobjekte funktionieren wie elektrische Schutzschalter, die den Strom automatisch unterbrechen, wenn im Stromkreis eine Störung auftritt. Elektrische Schutzschalter schalten den Stromfluss ab oder lösen ihn aus, wenn eine Störung vorliegt. In ähnlicher Weise befindet sich das Schutzschalter-Objekt zwischen dem Anrufer und dem angerufenen Dienst und löst aus, wenn der Anrufer nicht verfügbar ist.

Die Irrtümer des verteilten Rechnens sind eine Reihe von Behauptungen, die von Peter Deutsch und anderen Mitarbeitern von Sun Microsystems aufgestellt wurden. Sie sagen, dass Programmierer, die mit verteilten Anwendungen noch nicht vertraut sind, ausnahmslos falsche Annahmen treffen. Die Netzwerkzuverlässigkeit, die Erwartung, keine Latenz zu haben und Bandbreitenbeschränkungen führen dazu, dass Softwareanwendungen mit minimaler Fehlerbehandlung für Netzwerkfehler geschrieben werden.

Während eines Netzwerkausfalls warten Anwendungen möglicherweise auf unbestimmte Zeit auf eine Antwort und verbrauchen kontinuierlich Anwendungsressourcen. Wenn die Vorgänge nicht wiederholt werden, wenn das Netzwerk wieder verfügbar ist, kann dies ebenfalls zu einer Verschlechterung der Anwendung führen. Wenn bei API Aufrufen einer Datenbank oder eines externen Dienstes aufgrund von Netzwerkproblemen ein Timeout auftritt, können wiederholte Anrufe ohne Schutzschalter Kosten und Leistung beeinträchtigen.

Anwendbarkeit

Verwenden Sie dieses Muster, wenn:

- Der Anruferdienst tätigt einen Anruf, der höchstwahrscheinlich fehlschlagen wird.
- Eine hohe Latenz, die der angerufene Dienst aufweist (z. B. wenn die Datenbankverbindungen langsam sind), führt zu Timeouts beim angerufenen Dienst.
- Der Anruferdienst tätigt einen synchronen Anruf, aber der angerufene Dienst ist nicht verfügbar oder weist eine hohe Latenz auf.

Fehler und Überlegungen

- Dienstunabhängige Implementierung: Um zu verhindern, dass der Code aufgebläht wird, empfehlen wir, das Circuit-Breaker-Objekt auf Microservice-unabhängige und -gesteuerte Weise zu implementieren. API
- Schließung des Stromkreises durch den Angerufenen: Wenn der Angerufene das Leistungsproblem oder den Ausfall behoben hat, kann er den Verbindungsstatus auf ändern. `CLOSED` Dies ist eine Erweiterung des Schutzschaltermusters und kann implementiert werden, wenn Ihr Wiederherstellungszeitziel (RTO) dies erfordert.
- Multithread-Anrufe: Der Wert für das Ablauf-Timeout ist definiert als der Zeitraum, für den die Verbindung ausgelöst bleibt, bevor Anrufe erneut weitergeleitet werden, um die Verfügbarkeit des Dienstes zu überprüfen. Wenn der angerufene Dienst in mehreren Threads aufgerufen wird,

definiert der erste fehlgeschlagene Anruf den Wert für das Ablauftimeout. Ihre Implementierung sollte sicherstellen, dass bei nachfolgenden Aufrufen das Ablauftimeout nicht endlos verschoben wird.

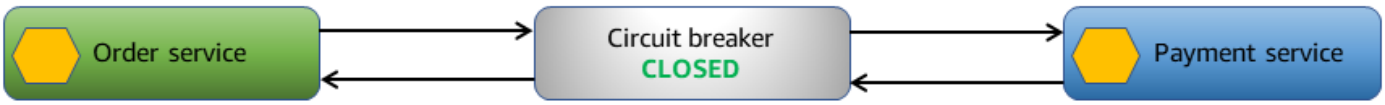
- Erzwungenes Öffnen oder Schließen des Stromkreises: Systemadministratoren sollten in der Lage sein, einen Kreislauf zu öffnen oder zu schließen. Dies kann erreicht werden, indem der Wert für das Ablauftimeout in der Datenbanktabelle aktualisiert wird.
- Beobachtbarkeit: In der Anwendung sollte eine Protokollierung eingerichtet sein, um die Anrufe zu identifizieren, die fehlschlagen, wenn der Schutzschalter geöffnet ist.

Implementierung

Hochrangige Architektur

Im folgenden Beispiel ist der Anrufer der Bestelldienst und der Angerufene der Zahlungsdienst.

Liegen keine Ausfälle vor, leitet der Bestellservice alle Anrufe über den Schutzschalter an den Zahlungsdienst weiter, wie das folgende Diagramm zeigt.



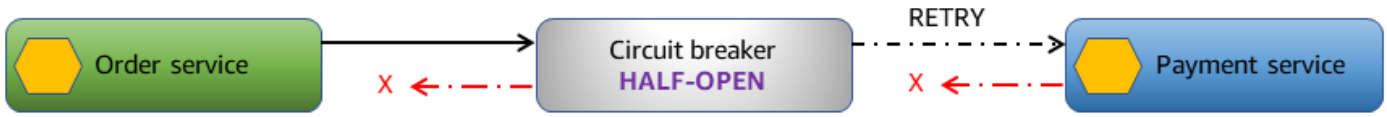
Wenn beim Zahlungsdienst ein Timeout auftritt, kann der Schutzschalter den Timeout erkennen und den Ausfall nachverfolgen.

Circuit breaker with payment service failure

Wenn die Timeouts einen bestimmten Schwellenwert überschreiten, öffnet die Anwendung den Stromkreis. Wenn der Stromkreis geöffnet ist, leitet das Circuit Breaker-Objekt die Anrufe nicht an den Zahlungsdienst weiter. Wenn der Bestellservice den Zahlungsdienst aufruft, wird sofort ein Fehler gemeldet.

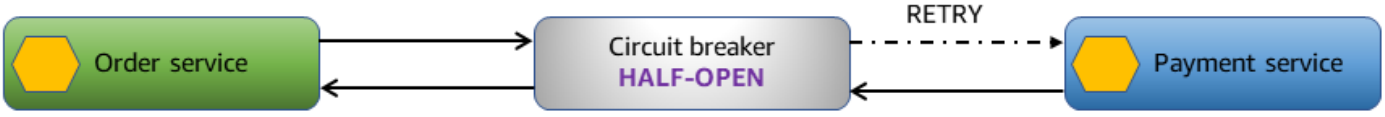
Circuit breaker stops routing to payment service

Das Circuit Breaker-Objekt versucht in regelmäßigen Abständen festzustellen, ob die Aufrufe an den Zahlungsdienst erfolgreich waren.



Circuit breaker periodically retries payment service

Wenn der Anruf beim Zahlungsdienst erfolgreich ist, wird die Verbindung geschlossen und alle weiteren Anrufe werden wieder an den Zahlungsdienst weitergeleitet.



Circuit breaker with working payment service

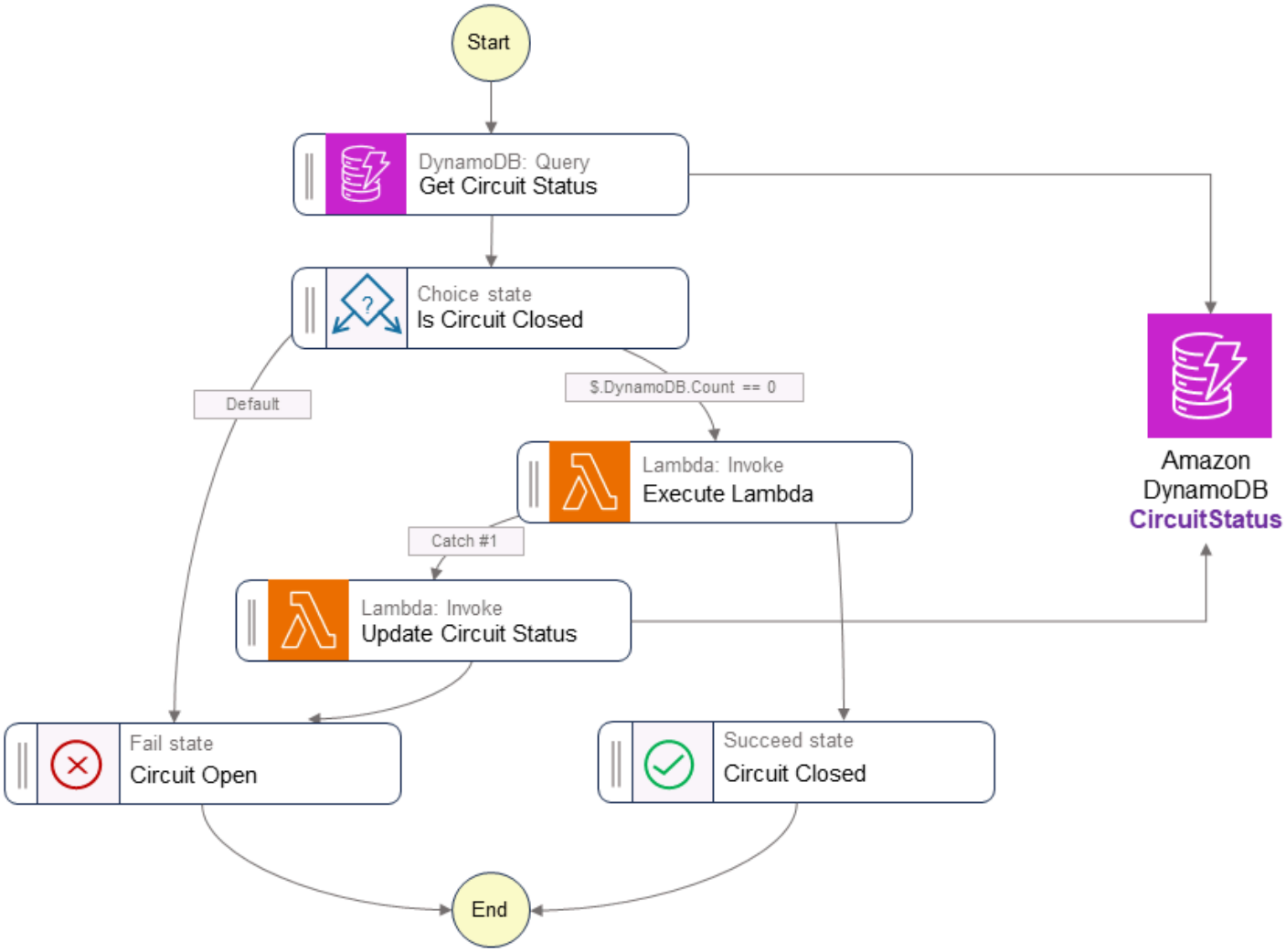
Implementierung mithilfe von AWS Diensten

Die Beispiellösung verwendet Express-Workflows AWS Step Functions, um das Circuit Breaker Pattern zu implementieren. Mit der Step Functions Functions-Zustandsmaschine können Sie die Wiederholungsfunktionen und den entscheidungsbasierten Kontrollfluss konfigurieren, die für die Musterimplementierung erforderlich sind.

Die Lösung verwendet auch eine Amazon DynamoDB-Tabelle als Datenspeicher, um den Schaltkreisstatus zu verfolgen. Dieser kann für eine bessere Leistung durch einen In-Memory-Datenspeicher wie Amazon ElastiCache (RedisOSS) ersetzt werden.

Wenn ein Dienst einen anderen Dienst aufrufen möchte, startet er den Workflow mit dem Namen des aufgerufenen Dienstes. Der Workflow ruft den Circuit Breaker-Status aus der `CircuitStatus` DynamoDB-Tabelle ab, in der die aktuell heruntergestuften Dienste gespeichert sind. `CircuitStatus`Enthält er einen noch nicht abgelaufenen Datensatz für den Angerufenen, ist der Circuit geöffnet. Der Step Functions Functions-Workflow gibt einen sofortigen Fehler zurück und wird mit einem `FAIL` Status beendet.

Wenn die `CircuitStatus` Tabelle keinen Datensatz für den Angerufenen oder einen abgelaufenen Datensatz enthält, ist der Dienst betriebsbereit. Der `ExecuteLambda` Schritt in der State-Machine-Definition ruft die Lambda-Funktion auf, die über einen Parameterwert gesendet wird. Wenn der Aufruf erfolgreich ist, wird der Step Functions Functions-Workflow mit einem `SUCCESS` Status beendet.



Schlägt der Serviceaufruf fehl oder tritt ein Timeout auf, versucht die Anwendung es mit exponentiellem Backoff für eine bestimmte Anzahl von Malen erneut. Schlägt der Serviceaufruf nach den Wiederholungen fehl, fügt der Workflow einen Datensatz mit dem Zeichen an in die `CircuitStatus` Tabelle für den Dienst ein`ExpiryTimeStamp`, und der Workflow wird mit einem Status beendet. `FAIL` Bei nachfolgenden Aufrufen desselben Dienstes wird sofort ein Fehler gemeldet, sofern der Schutzschalter geöffnet ist. Der `Get Circuit Status` Schritt in der State-Machine-Definition überprüft die Verfügbarkeit des Dienstes anhand des `ExpiryTimeStamp` Werts. Abgelaufene Elemente werden mithilfe der DynamoDB-Funktion time to live (TTL) aus der `CircuitStatus` Tabelle gelöscht.

Beispiel-Code

Der folgende Code verwendet die `GetCircuitStatus` Lambda-Funktion, um den Status des Leistungsschalters zu überprüfen.

```
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 = "";
}
```

Der folgende Code zeigt die Amazon States-Sprachanweisungen im Step Functions Functions-Workflow.

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

GitHub Repository

Eine vollständige Implementierung der Beispielarchitektur für dieses Muster finden Sie im GitHub Repository unter https://github.com/aws-samples/circuit-breaker-netcore-blog.

Blog-Referenzen

- Verwenden des Circuit Breaker Patterns mit AWS Step Functions Amazon DynamoDB

Verwandter Inhalt

- Würger-Feigenmuster
- Wiederholungsversuch mit Backoff-Muster
- AWS App Mesh Funktionen von Leistungsschaltern

Ereignis-Sourcing-Muster

Absicht

In ereignisgesteuerten Architekturen speichert das Ereignis-Sourcing-Muster die Ereignisse, die zu einer Statusänderung führen, in einem Datenspeicher. Dies hilft dabei, eine vollständige Historie von Statusänderungen zu erfassen und zu verwalten, und fördert die Überprüfbarkeit, Rückverfolgbarkeit und die Fähigkeit, vergangene Zustände zu analysieren.

Motivation

Mehrere Microservices können zusammenarbeiten, um Anfragen zu bearbeiten, und sie kommunizieren über Ereignisse. Diese Ereignisse können zu einer Änderung des Zustands (der Daten) führen. Das Speichern von Ereignisobjekten in der Reihenfolge ihres Auftretens liefert wertvolle Informationen über den aktuellen Zustand der Dateneinheit und zusätzliche Informationen darüber, wie sie zu diesem Zustand gekommen ist.

Anwendbarkeit

Verwenden Sie das Ereignis-Sourcing-Muster, wenn:

- Für die Nachverfolgung ein unveränderlicher Verlauf der Ereignisse, die in einer Anwendung auftreten, erforderlich ist.
- Polyglotte Datenprojektionen aus einer Single Source of Truth (SSOT) benötigt werden.
- Eine zeitpunktgenaue Rekonstruktion des Anwendungsstatus erforderlich ist.
- Eine langfristige Speicherung des Anwendungsstatus nicht erforderlich ist, aber Sie ihn bei Bedarf rekonstruieren möchten.
- Workloads unterschiedliche Lese- und Schreibvolumen haben. Sie haben beispielsweise schreibintensive Workloads, für die keine Echtzeitverarbeitung erforderlich ist.
- Change Data Capture (CDC) erforderlich ist, um die Anwendungsleistung und andere Metriken zu analysieren.
- Audit-Daten für alle Ereignisse, die in einem System auftreten, für Berichts- und Compliance-Zwecke erforderlich sind.

- Sie Was-wäre-wenn-Szenarien ableiten möchten, indem Sie während des Wiedergabeprozesses Ereignisse ändern (einfügen, aktualisieren oder löschen), um den möglichen Endzustand zu bestimmen.

Fehler und Überlegungen

- Optimistic Concurrency Control: Dieses Muster speichert jedes Ereignis, das eine Statusänderung im System verursacht. Mehrere Benutzer oder Services können versuchen, dieselben Daten gleichzeitig zu aktualisieren, was zu Ereigniskonflikten führen kann. Diese Konflikte treten auf, wenn widersprüchliche Ereignisse gleichzeitig erzeugt und angewendet werden, was zu einem endgültigen Datenstatus führt, der nicht der Realität entspricht. Um dieses Problem zu beheben, können Sie Strategien zum Erkennen und Auflösen von Ereigniskonflikten implementieren. Sie können beispielsweise ein Schema für Optimistic Concurrency Control implementieren, indem Sie eine Versionsverwaltung hinzufügen oder Ereignisse mit Zeitstempeln versehen, um die Reihenfolge der Aktualisierungen zu verfolgen.
- Komplexität: Die Implementierung von Event Sourcing erfordert ein Umdenken von traditionellen CRUD-Operationen hin zu ereignisgesteuertem Denken. Der Wiedergabeprozess, mit dem der ursprüngliche Zustand des Systems wiederhergestellt wird, kann komplex sein, um die Idempotenz der Daten zu gewährleisten. Auch die Speicherung von Ereignissen, Backups und Snapshots kann die Komplexität erhöhen.
- Ereigniskonsistenz: Die aus den Ereignissen abgeleiteten Datenprojektionen sind aufgrund der Latenz bei der Aktualisierung der Daten durch das Muster Command Query Responsibility Segregation (CQRS, Segmentierung der Zuständigkeiten bei Befehlsabfragen) oder materialisierte Ansicht konsistent hinsichtlich der Ereignissen. Wenn Verbraucher Daten aus einem Ereignisspeicher verarbeiten und Publisher neue Daten senden, kann es sein, dass die Datenprojektion oder das Anwendungsobjekt nicht den aktuellen Zustand wiedergibt.
- Abfragen: Das Abrufen von aktuellen oder aggregierten Daten aus Ereignisprotokollen kann im Vergleich zu herkömmlichen Datenbanken aufwändiger und langsamer sein, insbesondere bei komplexen Abfragen und Berichtsaufgaben. Um dieses Problem zu beheben, wird Ereignis-Sourcing häufig mit dem CQRS-Muster implementiert.
- Größe und Kosten des Ereignisspeichers: Die Größe des Ereignisspeichers kann exponentiell ansteigen, da die Ereignisse kontinuierlich aufbewahrt werden, insbesondere in Systemen mit hohem Ereignisdurchsatz oder langen Aufbewahrungszeiten. Daher müssen Sie die Ereignisdaten regelmäßig auf kostengünstigem Speicher archivieren, um zu verhindern, dass der Ereignisspeicher zu groß wird.

- Skalierbarkeit des Ereignisspeichers: Der Ereignisspeicher muss große Mengen an Schreib- und Lesevorgängen effizient verarbeiten. Das Skalieren eines Ereignisspeichers kann eine Herausforderung sein. Daher ist es wichtig, über einen Datenspeicher zu verfügen, der Shards und Partitionen bereitstellt.
- Effizienz und Optimierung: Wählen oder entwerfen Sie einen Ereignisspeicher, der sowohl Schreib- als auch Lesevorgänge effizient verarbeitet. Der Ereignisspeicher sollte für das erwartete Ereignisvolumen und die Abfragemuster für die Anwendung optimiert werden. Durch die Implementierung von Indizierungs- und Abfragemechanismen kann der Abruf von Ereignissen bei der Rekonstruktion des Anwendungsstatus beschleunigt werden. Sie können auch die Verwendung spezialisierter Datenbanken oder Bibliotheken für Ereignisspeicher in Betracht ziehen, die Features zur Abfrageoptimierung bieten.
- Snapshots: Sie müssen die Ereignisprotokolle in regelmäßigen Abständen mit zeitbasierter Aktivierung sichern. Die Wiedergabe der Ereignisse auf dem letzten bekannten erfolgreichen Backup der Daten sollte zu einer zeitpunktgenauen Wiederherstellung des Anwendungsstatus führen. Das Recovery Point Objective (RPO) ist die maximal zulässige Zeitspanne seit dem letzten Datenwiederherstellungspunkt. RPO bestimmt, was als akzeptabler Datenverlust zwischen dem letzten Wiederherstellungspunkt und der Serviceunterbrechung angesehen wird. Die Häufigkeit der täglichen Snapshots des Daten- und Ereignisspeichers sollte auf dem RPO für die Anwendung basieren.
- Zeitabhängigkeit: Die Ereignisse werden in der Reihenfolge gespeichert, in der sie auftreten. Daher ist die Netzwerkzuverlässigkeit ein wichtiger Faktor, den Sie bei der Implementierung dieses Musters berücksichtigen sollten. Latenzprobleme können zu einem fehlerhaften Systemstatus führen. Verwenden Sie First-in, First-out (FIFO)-Warteschlangen mit maximal einmaliger Zustellung, um die Ereignisse in den Ereignisspeicher zu übertragen.
- Ereignis-Wiedergabeleistung: Die Wiedergabe einer großen Anzahl von Ereignissen zur Rekonstruktion des aktuellen Anwendungsstatus kann zeitaufwändig sein. Es sind Optimierungsmaßnahmen erforderlich, um die Leistung zu verbessern, insbesondere bei der Wiedergabe von Ereignissen aus archivierten Daten.
- Externe Systemaktualisierungen: Anwendungen, die das Event-Sourcing-Muster verwenden, aktualisieren möglicherweise Datenspeicher in externen Systemen und erfassen diese Aktualisierungen möglicherweise als Ereignisobjekte. Bei der Wiedergabe von Ereignissen kann dies zu einem Problem werden, wenn das externe System keine Aktualisierung erwartet. In solchen Fällen können Sie Feature-Flags verwenden, um externe Systemaktualisierungen zu steuern.

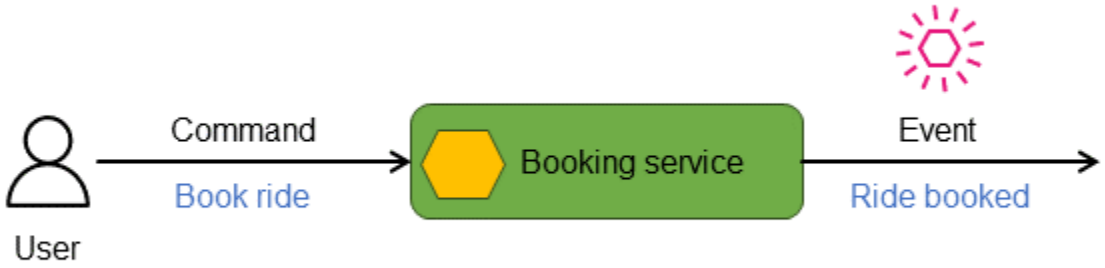
- Externe Systemabfragen: Wenn externe Systemaufrufe auf das Datum und die Uhrzeit des Aufrufs reagieren, können die empfangenen Daten in internen Datenspeichern gespeichert werden, um sie bei Wiedergaben zu verwenden.
- Ereignis-Versionsverwaltung: Mit der Weiterentwicklung der Anwendung kann sich die Struktur der Ereignisse (Schema) ändern. Es ist notwendig, eine Versionsverwaltungsstrategie für Ereignisse zu implementieren, um die Rückwärts- und Vorwärtskompatibilität sicherzustellen. Dies kann die Aufnahme eines Versionsfeldes in die Ereignisnutzlast und die angemessene Behandlung verschiedener Ereignisversionen während der Wiedergabe beinhalten.

Implementierung

Hochrangige Architektur

Befehle und Ereignisse

In verteilten, ereignisgesteuerten Microservice-Anwendungen stellen Befehle die Anweisungen oder Anfragen dar, die an einen Service gesendet werden, in der Regel mit der Absicht, eine Änderung seines Zustands einzuleiten. Der Service verarbeitet diese Befehle und bewertet die Gültigkeit und Anwendbarkeit des Befehls auf seinen aktuellen Status. Wenn der Befehl erfolgreich ausgeführt wird, reagiert der Service, indem er ein Ereignis ausgibt, das die ergriffene Aktion und die entsprechenden Statusinformationen angibt. In der folgenden Abbildung reagiert der Buchungsservice beispielsweise auf den Befehl „Fahrt buchen“, indem er das Ereignis „Fahrt gebucht“ ausgibt.



Ereignis-Speicher

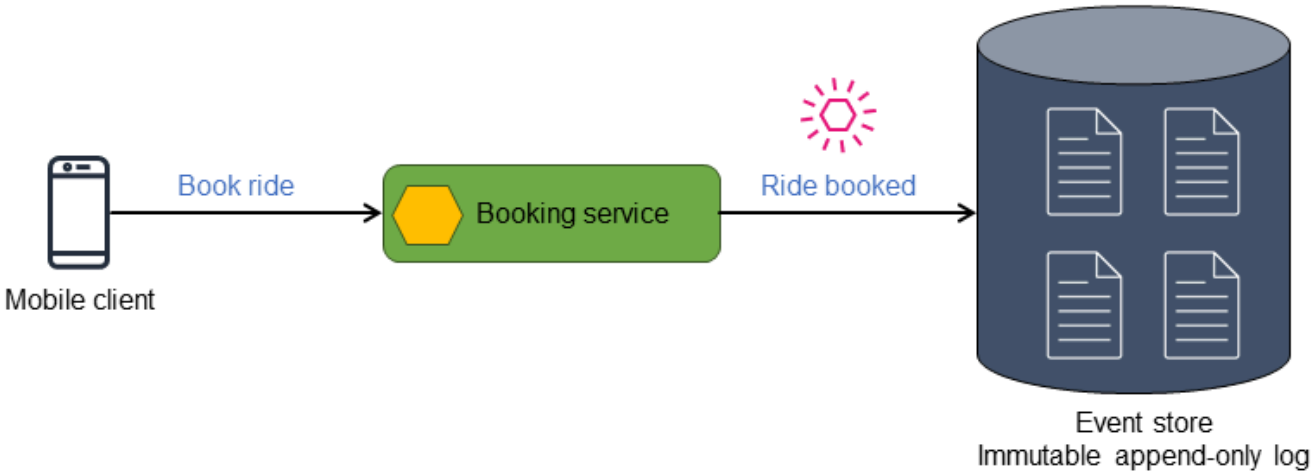
Ereignisse werden in einem unveränderlichen und chronologisch geordneten Append-Only-Repository oder -Datenspeicher, dem sogenannten Ereignis-Speicher, protokolliert. Jede Statusänderung wird als individuelles Ereignisobjekt behandelt. Ein Entitätsobjekt oder ein

Datenspeicher mit bekanntem Anfangszustand, seinem aktuellen Zustand und einer beliebigen zeitpunktbezogenen Ansicht kann rekonstruiert werden, indem die Ereignisse in der Reihenfolge ihres Auftretens wiedergegeben werden.

Der Ereignis-Speicher dient als historische Aufzeichnung aller Aktionen und Statusänderungen und dient als wertvolle Single Source of Truth. Sie können den Ereignis-Speicher verwenden, um den endgültigen, aktuellen Status des Systems abzuleiten, indem Sie die Ereignisse über einen Wiedergabeprozessor weiterleiten, der diese Ereignisse anwendet, um eine genaue Darstellung des aktuellen Systemstatus zu erstellen. Sie können den Ereignis-Speicher auch verwenden, um den Status zu einem bestimmten Zeitpunkt darzustellen, indem Sie die Ereignisse über einen Wiedergabeprozessor wiedergeben. Im Ereignis-Sourcing-Muster wird der aktuelle Status möglicherweise nicht vollständig durch das jüngste Ereignisobjekt repräsentiert. Sie können den aktuellen Status auf drei Arten ableiten:

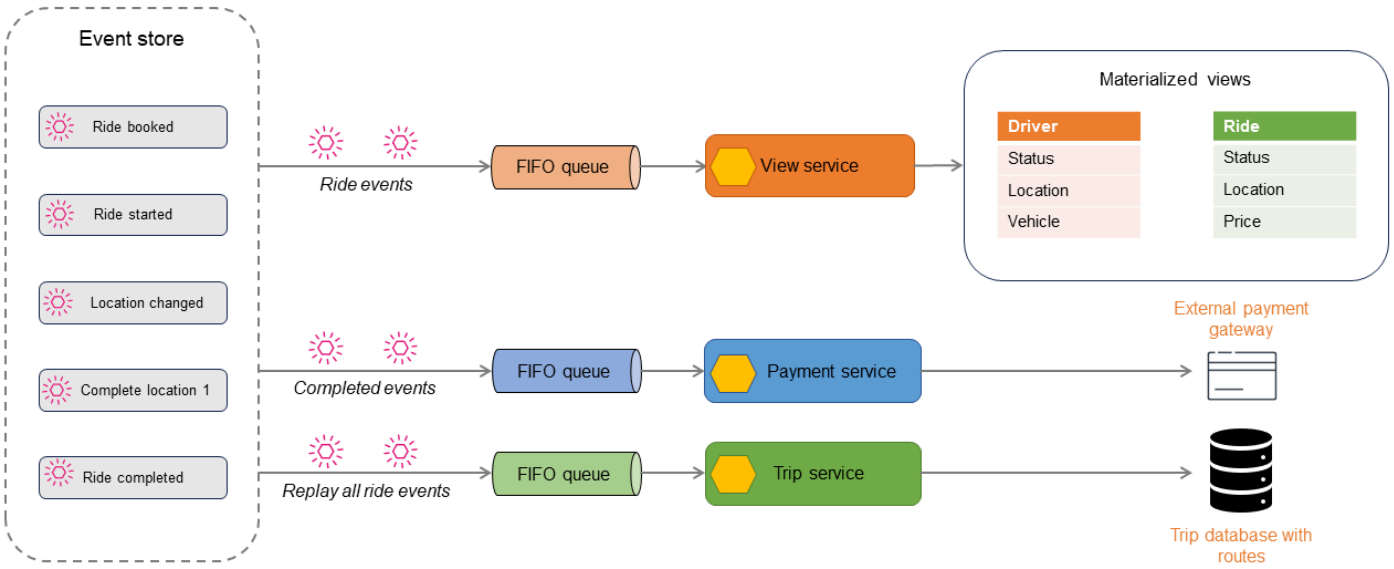
- Durch die Aggregation verwandter Ereignisse. Die zugehörigen Ereignisobjekte werden kombiniert, um den aktuellen Status für die Abfrage zu generieren. Dieser Ansatz wird häufig in Verbindung mit dem CQRS-Muster verwendet, bei dem die Ereignisse kombiniert und in den schreibgeschützten Datenspeicher geschrieben werden.
- Durch das Verwenden der materialisierten Ansicht. Sie können Event Sourcing mit dem Materialisierte-Ansicht-Muster verwenden, um die Ereignisdaten zu berechnen oder zusammenzufassen und den aktuellen Status der zugehörigen Daten zu ermitteln.
- Durch die Wiedergabe von Ereignissen. Ereignisobjekte können wiedergegeben werden, um Aktionen zur Generierung des aktuellen Status auszuführen.

Das folgende Diagramm zeigt, wie das Ereignis `Ride booked` in einem Ereignisspeicher gespeichert wird.



Der Ereignisspeicher veröffentlicht die Ereignisse, die er speichert, und die Ereignisse können gefiltert und für nachfolgende Aktionen an den entsprechenden Prozessor weitergeleitet werden. Ereignisse können beispielsweise an einen Anzeigeprozessor weitergeleitet werden, der den Status zusammenfasst und eine materialisierte Ansicht anzeigt. Die Ereignisse werden in das Datenformat des Zieldatenspeichers umgewandelt. Diese Architektur kann erweitert werden, um verschiedene Arten von Datenspeichern abzuleiten, was zu einer polyglotten Persistenz der Daten führt.

Das folgende Diagramm beschreibt die Ereignisse in einer Anwendung zur Buchung von Fahrten. Alle Ereignisse, die innerhalb der Anwendung auftreten, werden im Ereignisspeicher gespeichert. Gespeicherte Ereignisse werden dann gefiltert und an verschiedene Verbraucher weitergeleitet.



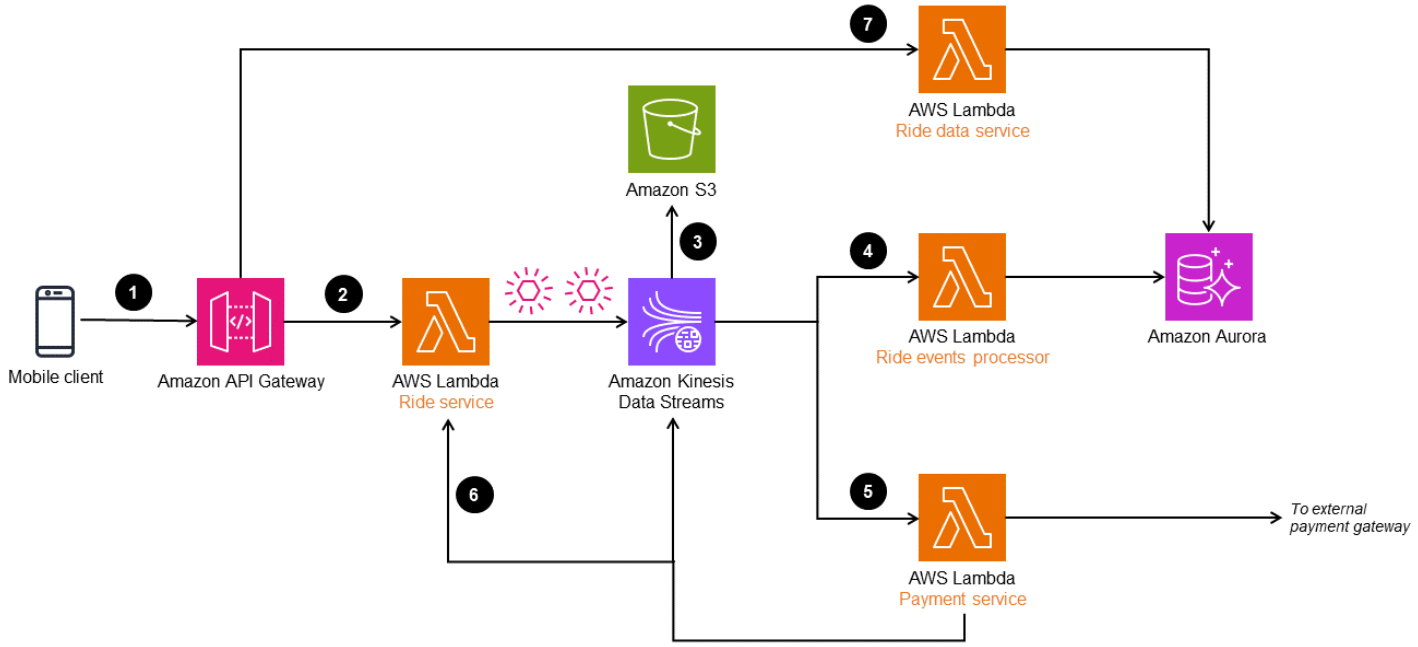
Die Fahrt-Ereignisse können verwendet werden, um schreibgeschützte Datenspeicher mithilfe der CQRS- oder Materialisierte-Ansicht-Muster zu generieren. Sie können den aktuellen Status der Fahrt, des Fahrers oder der Buchung erfahren, indem Sie die Lesespeicher abfragen. Einige Ereignisse, wie z. B. `Location changed` oder `Ride completed`, werden an einen anderen Verbraucher zur Zahlungsabwicklung veröffentlicht. Wenn die Fahrt beendet ist, werden alle Ereignisse der Fahrt wiedergegeben, um einen Verlauf der Fahrt für Prüf- oder Berichtszwecke zu erstellen.

Das Ereignis-Sourcing-Muster wird häufig in Anwendungen verwendet, die eine zeitpunktbezogene Wiederherstellung erfordern, und auch dann, wenn die Daten mithilfe einer Single Source of Truth in verschiedenen Formaten projiziert werden müssen. Beide Vorgänge erfordern einen Wiedergabeprozess, um die Ereignisse auszuführen und den erforderlichen Endzustand abzuleiten. Der Wiedergabeprozessor könnte auch einen bekannten Startpunkt benötigen – idealerweise nicht beim Start der Anwendung, denn das würde keinen effizienten Prozess darstellen. Wir empfehlen Ihnen, regelmäßig Snapshots des Systemstatus zu erstellen und eine kleinere Anzahl von Ereignissen anzuwenden, um einen aktuellen Status zu erhalten.

Implementierung mithilfe von AWS-Services

In der folgenden Architektur wird Amazon Kinesis Data Streams als Ereignisspeicher verwendet. Dieser Service erfasst und verwaltet Anwendungsänderungen als Ereignisse und bietet eine Daten-Streaming-Lösung mit hohem Durchsatz und in Echtzeit. Um das Ereignis-Sourcing-Muster in AWS zu implementieren, können Sie je nach den Anforderungen Ihrer Anwendung auch Services wie Amazon EventBridge und Amazon Managed Streaming für Apache Kafka (Amazon MSK) verwenden.

Um die Haltbarkeit zu erhöhen und Prüfungen zu ermöglichen, können Sie die von Kinesis Data Streams erfassten Ereignisse in Amazon Simple Storage Service (Amazon S3) archivieren. Dieser duale Speicheransatz hilft, historische Ereignisdaten für zukünftige Analysen und Compliance-Zwecke sicher beizubehalten.



Der Workflow besteht aus folgenden Schritten:

1. Eine Fahrtbuchungsanfrage wird über einen mobilen Client an einen Amazon-API-Gateway-Endpunkt gestellt.
2. Der Fahrt-Microservice (`Ride service`-Lambda-Funktion) empfängt die Anfrage, wandelt die Objekte um und veröffentlicht sie in Kinesis Data Streams.
3. Die Ereignisdaten in Kinesis Data Streams werden zu Compliance- und Prüfungs-Zwecken in Amazon S3 gespeichert.
4. Die Ereignisse werden von der `Ride event processor`-Lambda-Funktion umgewandelt und verarbeitet und in einer Amazon-Aurora-Datenbank gespeichert, um eine materialisierte Ansicht für die Fahrtdaten bereitzustellen.
5. Die abgeschlossenen Fahrt-Ereignisse werden gefiltert und zur Zahlungsabwicklung an ein externes Zahlungs-Gateway gesendet. Wenn die Zahlung abgeschlossen ist, wird ein weiteres Ereignis an Kinesis Data Streams gesendet, um die Fahrt-Datenbank zu aktualisieren.
6. Wenn die Fahrt abgeschlossen ist, werden die Fahrt-Ereignisse an die `Ride service`-Lambda-Funktion wiedergegeben, um Routen und den Verlauf der Fahrt zu erstellen.

7. Fahrt-Informationen können über den `Ride data service`, der aus der Aurora-Datenbank gelesen wird, abgerufen werden.

API Gateway kann auch ohne die `Ride service`-Lambda-Funktion das Ereignisobjekt direkt an Kinesis Data Streams senden. In einem komplexen System wie einem Taxidienst muss das Ereignisobjekt jedoch möglicherweise verarbeitet und angereichert werden, bevor es in den Datenstrom aufgenommen wird. Aus diesem Grund verfügt die Architektur über einen `Ride service`, der das Ereignis verarbeitet, bevor er es an Kinesis Data Streams sendet.

Blog-Referenzen

- Neu für AWS Lambda – SQS FIFO als Ereignisquelle

Sechseckiges Architekturmuster

Absicht

Das hexagonale Architekturmuster, auch bekannt als Port- und Adaptermuster, wurde 2005 von Dr. Alistair Cockburn vorgeschlagen. Es zielt darauf ab, lose gekoppelte Architekturen zu schaffen, in denen Anwendungskomponenten unabhängig voneinander getestet werden können, ohne von Datenspeichern oder Benutzeroberflächen (UIs) abhängig zu sein. Dieses Muster trägt dazu bei, dass Datenspeicher und Benutzeroberflächen nicht an bestimmte Technologien gebunden sind. Dies macht es einfacher, den Technologie-Stack im Laufe der Zeit zu ändern, mit begrenzten oder keinen Auswirkungen auf die Geschäftslogik. In dieser lose gekoppelten Architektur kommuniziert die Anwendung mit externen Komponenten über Schnittstellen, die als Ports bezeichnet werden, und verwendet Adapter, um den technischen Austausch mit diesen Komponenten zu übersetzen.

Motivation

Das hexagonale Architekturmuster wird verwendet, um die Geschäftslogik (Domänenlogik) vom zugehörigen Infrastrukturcode zu isolieren, z. B. vom Code für den Zugriff auf eine Datenbank oder externe APIs. Dieses Muster ist nützlich, um lose gekoppelte Geschäftslogik und Infrastrukturcode für AWS Lambda Funktionen zu erstellen, die eine Integration mit externen Diensten erfordern. In herkömmlichen Architekturen ist es üblich, Geschäftslogik in Form von gespeicherten Prozeduren in die Datenbankebene und in die Benutzeroberfläche einzubetten. Diese Vorgehensweise führt zusammen mit der Verwendung von UI-spezifischen Konstrukten innerhalb der Geschäftslogik zu eng miteinander verknüpften Architekturen, die zu Engpässen bei Datenbankmigrationen und der Modernisierung der Benutzererfahrung (UX) führen. Das hexagonale Architekturmuster ermöglicht es Ihnen, Ihre Systeme und Anwendungen nach Zweck und nicht nach Technologie zu entwerfen. Diese Strategie führt zu leicht austauschbaren Anwendungskomponenten wie Datenbanken, UX und Servicekomponenten.

Anwendbarkeit

Verwenden Sie das hexagonale Architekturmuster, wenn:

- Sie möchten Ihre Anwendungsarchitektur entkoppeln, um Komponenten zu erstellen, die vollständig getestet werden können.
- Mehrere Clienttypen können dieselbe Domänenlogik verwenden.

- Ihre Benutzeroberflächen- und Datenbankkomponenten erfordern regelmäßige Technologieaktualisierungen, die sich nicht auf die Anwendungslogik auswirken.
- Ihre Anwendung erfordert mehrere Eingabeanbieter und Ausgabeverbraucher, und die Anpassung der Anwendungslogik führt zu Komplexität des Codes und mangelnder Erweiterbarkeit.

Fehler und Überlegungen

- Domänengesteuertes Design: Die hexagonale Architektur eignet sich besonders gut für domänengesteuertes Design (DDD). Jede Anwendungskomponente stellt eine Unterdomäne in DDD dar, und hexagonale Architekturen können verwendet werden, um eine lose Kopplung zwischen Anwendungskomponenten zu erreichen.
- Testbarkeit: Eine hexagonale Architektur verwendet konstruktionsbedingt Abstraktionen für Eingaben und Ausgaben. Daher wird es aufgrund der inhärenten losen Kopplung einfacher, Unit-Tests zu schreiben und isoliert zu testen.
- Komplexität: Die Komplexität der Trennung von Geschäftslogik und Infrastrukturcode kann bei sorgfältiger Handhabung große Vorteile wie Agilität, Testabdeckung und Anpassungsfähigkeit der Technologie mit sich bringen. Andernfalls kann es schwierig werden, Probleme zu lösen.
- Wartungsaufwand: Der zusätzliche Adaptercode, der die Architektur austauschbar macht, ist nur gerechtfertigt, wenn die Anwendungskomponente mehrere Eingabequellen und Ausgabeziele zum Schreiben benötigt oder wenn sich die Eingabe- und Ausgabedatenspeicher im Laufe der Zeit ändern müssen. Andernfalls wird der Adapter zu einer weiteren zusätzlichen Ebene, die gewartet werden muss, was zu einem Wartungsaufwand führt.
- Latenzprobleme: Durch die Verwendung von Anschlüssen und Adaptern wird eine weitere Ebene hinzugefügt, was zu Latenz führen kann.

Implementierung

Hexagonale Architekturen unterstützen die Isolierung von Anwendungs- und Geschäftslogik vom Infrastrukturcode und vom Code, der die Anwendung mit Benutzeroberflächen, externen APIs, Datenbanken und Message Brokern integriert. Sie können Geschäftslogikkomponenten einfach über Ports und Adapter mit anderen Komponenten (wie Datenbanken) in der Anwendungsarchitektur verbinden.

Ports sind technologieunabhängige Einstiegspunkte in eine Anwendungskomponente. Diese benutzerdefinierten Schnittstellen bestimmen die Schnittstelle, über die externe Akteure mit der

Anwendungskomponente kommunizieren können, unabhängig davon, wer oder was die Schnittstelle implementiert. Dies ähnelt der Art und Weise, wie ein USB-Anschluss vielen verschiedenen Gerätetypen die Kommunikation mit einem Computer ermöglicht, sofern sie einen USB-Adapter verwenden.

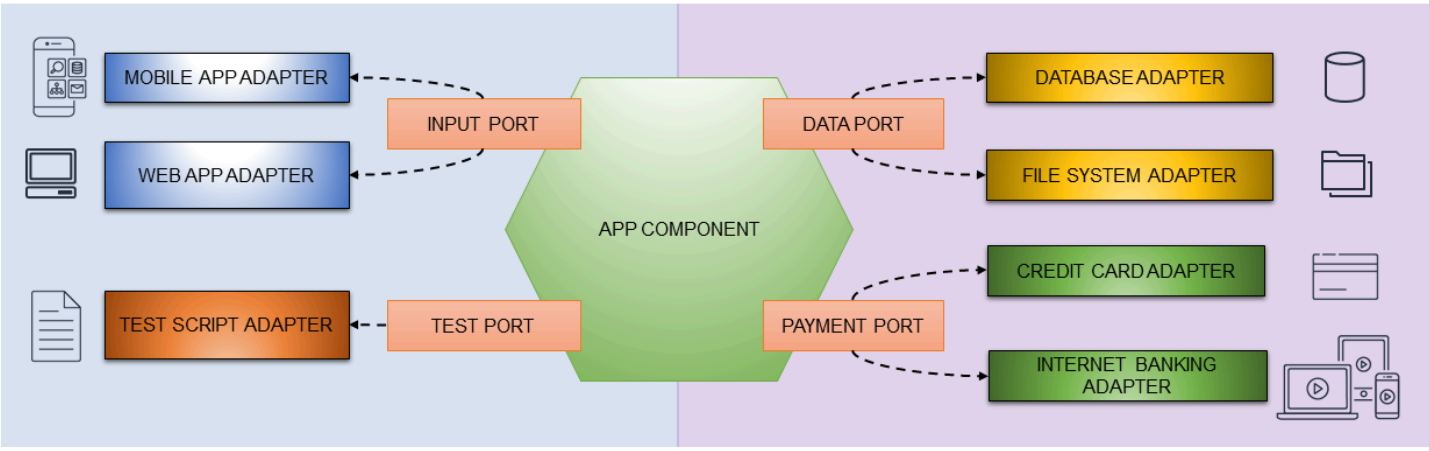
Adapter interagieren mithilfe einer bestimmten Technologie über einen Port mit der Anwendung. Adapter werden an diese Anschlüsse angeschlossen, empfangen Daten von den Anschlüssen oder stellen Daten an diese zur Verfügung und transformieren die Daten für die weitere Verarbeitung. Ein REST-Adapter ermöglicht es Akteuren beispielsweise, über eine REST-API mit der Anwendungskomponente zu kommunizieren. Ein Port kann mehrere Adapter haben, ohne dass ein Risiko für den Port oder die Anwendungskomponente besteht. Um das vorherige Beispiel zu erweitern, bietet das Hinzufügen eines GraphQL-Adapters zu demselben Port eine zusätzliche Möglichkeit für Akteure, über eine GraphQL-API mit der Anwendung zu interagieren, ohne die REST-API, den Port oder die Anwendung zu beeinträchtigen.

Ports stellen eine Verbindung zur Anwendung her, und Adapter dienen als Verbindung zur Außenwelt. Sie können Ports verwenden, um lose gekoppelte Anwendungskomponenten zu erstellen und abhängige Komponenten auszutauschen, indem Sie den Adapter ändern. Auf diese Weise kann die Anwendungskomponente mit externen Eingaben und Ausgaben interagieren, ohne dass sie über ein gewisses Maß an Kontextsensitivität verfügen muss. Die Komponenten sind auf jeder Ebene austauschbar, was automatisierte Tests erleichtert. Sie können Komponenten unabhängig voneinander testen, ohne vom Infrastrukturcode abhängig zu sein, anstatt eine gesamte Umgebung für die Durchführung von Tests bereitzustellen. Die Anwendungslogik hängt nicht von externen Faktoren ab, sodass das Testen vereinfacht wird und es einfacher wird, Abhängigkeiten nachzuahmen.

In einer lose gekoppelten Architektur sollte eine Anwendungskomponente beispielsweise in der Lage sein, Daten zu lesen und zu schreiben, ohne die Details des Datenspeichers zu kennen. Die Anwendungskomponente ist dafür verantwortlich, Daten an eine Schnittstelle (Port) zu liefern. Ein Adapter definiert die Logik des Schreibens in einen Datenspeicher, der je nach den Anforderungen der Anwendung eine Datenbank, ein Dateisystem oder ein Objektspeichersystem wie Amazon S3 sein kann.

Hochrangige Architektur

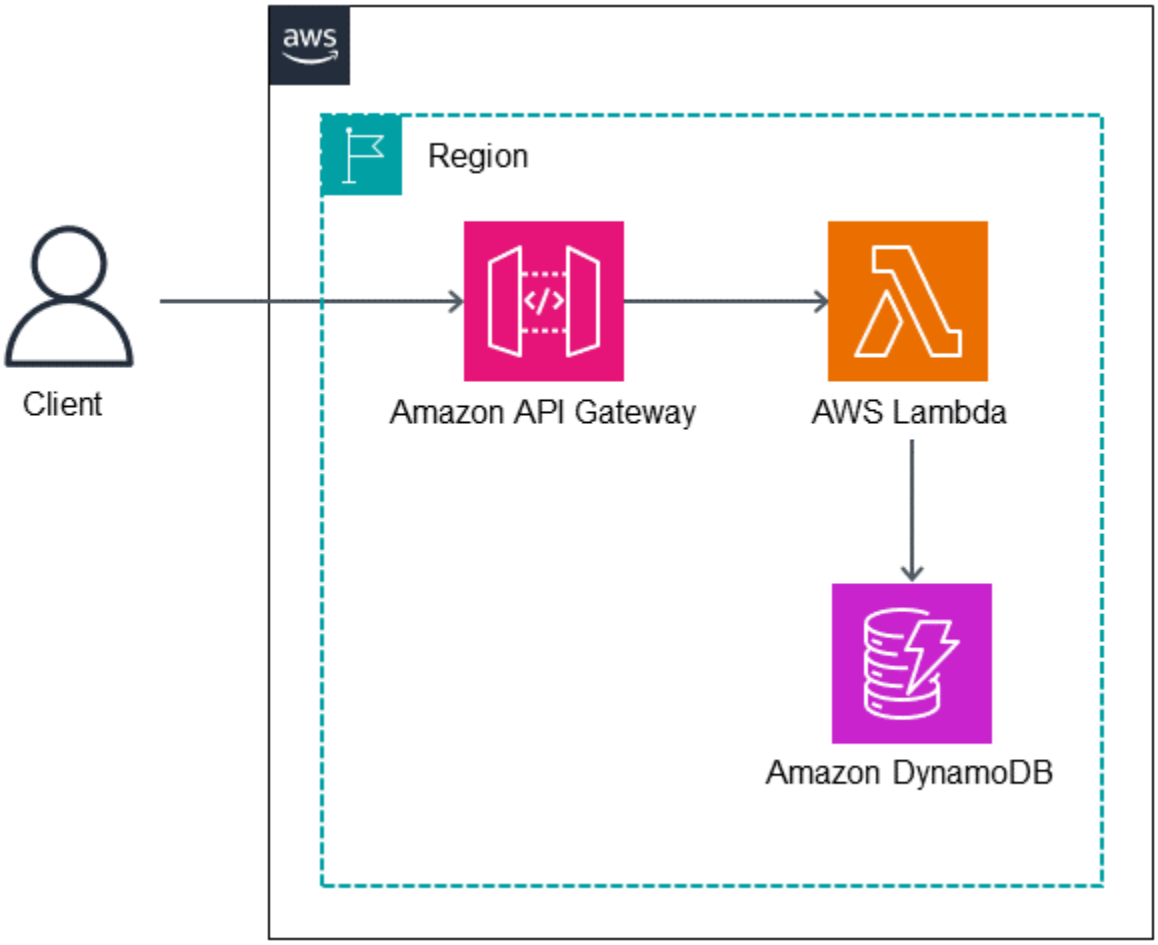
Die Anwendung oder Anwendungskomponente enthält die Kerngeschäftslogik. Sie empfängt Befehle oder Abfragen von den Ports und sendet Anfragen über die Ports an externe Akteure, die über Adapter implementiert werden, wie in der folgenden Abbildung dargestellt.



Implementierung mit AWS -Services

AWS Lambda Funktionen enthalten häufig sowohl Geschäftslogik als auch Datenbankintegrationscode, die eng miteinander verknüpft sind, um ein bestimmtes Ziel zu erreichen. Sie können das hexagonale Architekturmuster verwenden, um Geschäftslogik vom Infrastrukturcode zu trennen. Diese Trennung ermöglicht Komponententests der Geschäftslogik ohne Abhängigkeiten vom Datenbankcode und verbessert die Agilität des Entwicklungsprozesses.

In der folgenden Architektur implementiert eine Lambda-Funktion das hexagonale Architekturmuster. Die Lambda-Funktion wird von der Amazon API Gateway Gateway-REST-API initiiert. Die Funktion implementiert Geschäftslogik und schreibt Daten in DynamoDB-Tabellen.



Beispiel-Code

Der Beispielcode in diesem Abschnitt zeigt, wie Sie das Domänenmodell mithilfe von Lambda implementieren, es vom Infrastrukturcode (z. B. dem Code für den Zugriff auf DynamoDB) trennen und Komponententests für die Funktion implementieren.

Domänenmodell

Die Domänenmodellklasse kennt keine externen Komponenten oder Abhängigkeiten — sie implementiert lediglich die Geschäftslogik. Im folgenden Beispiel `Recipient` handelt es sich bei der Klasse um eine Domänenmodellklasse, die nach Überschneidungen beim Reservierungsdatum sucht.

```
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:
.....
```

Eingangsport

Die `RecipientInputPort` Klasse stellt eine Verbindung zur Empfängerklasse her und führt die Domänenlogik aus.

```
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

def make_reservation(self, recipient_id:str, slot_id:str) -> Status:
status = None

recipient = self.__recipient_output_port.get_recipient_by_id(recipient_id)
slot = self.__slot_output_port.get_slot_by_id(slot_id)
.....

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

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

DynamoDB-Adapterklasse

Die `DDBRecipientAdapter` Klasse implementiert den Zugriff auf die DynamoDB-Tabellen.

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

Die Lambda-Funktion `get_recipient_input_port` ist eine Fabrik für Instanzen der `RecipientInputPort` Klasse. Sie erstellt Instanzen von Ausgabeportklassen mit verwandten Adapterinstanzen.

```
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
}),
}
```

Komponententests

Sie können die Geschäftslogik für Domänenmodellklassen testen, indem Sie Scheinklassen einfügen. Das folgende Beispiel stellt den Komponententest für die `Recipent` Domänenmodellklasse bereit.

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

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

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

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

GitHub Repository

Eine vollständige Implementierung der Beispielarchitektur für dieses Muster finden Sie im GitHub Repository unter https://github.com/aws-samples/ aws-lambda-domain-model -sample.

Verwandter Inhalt

- Hexagonale Architektur, Artikel von Alistair Cockburn
- Entwicklung evolutionärer Architekturen mit AWS Lambda(Blogbeitrag auf Japanisch)AWS

Videos

Das folgende Video (auf Japanisch) beschreibt die Verwendung einer hexagonalen Architektur bei der Implementierung eines Domänenmodells mithilfe einer Lambda-Funktion.

Publish-Subscribe-Muster

Absicht

Das Publish-Subscribe-Muster, das auch als Pub-Sub-Muster bekannt ist, ist ein Nachrichtenmuster, das einen Nachrichtensender (Publisher) von interessierten Empfängern (Subscriber) entkoppelt. Dieses Muster implementiert eine asynchrone Kommunikation durch die Veröffentlichung von Nachrichten oder Ereignissen über einen Vermittler, den so genannten Message Broker oder Router (Nachrichteninfrastruktur). Das Publish-Subscribe-Muster erhöht die Skalierbarkeit und Reaktionsfähigkeit für Absender, indem es die Verantwortung für die Nachrichtenzustellung an die Nachrichteninfrastruktur abgibt, so dass sich der Absender auf die eigentliche Nachrichtenverarbeitung konzentrieren kann.

Motivation

In verteilten Architekturen müssen Systemkomponenten oft Informationen an andere Komponenten weitergeben, wenn Ereignisse innerhalb des Systems stattfinden. Das Publish-Subscribe-Muster trennt Belange, so dass sich Anwendungen auf ihre Kernfunktionen konzentrieren können, während die Nachrichteninfrastruktur Kommunikationsaufgaben wie das Routing von Nachrichten und die zuverlässige Zustellung übernimmt. Das Publish-Subscribe-Muster ermöglicht asynchrone Nachrichtenübermittlung, um Publisher und Subscriber zu entkoppeln. Publisher können Nachrichten auch ohne das Wissen der Subscriber versenden.

Anwendbarkeit

Verwenden Sie das Publish-Subscribe-Muster, wenn:

- Parallele Verarbeitung erforderlich ist, wenn eine einzelne Nachricht Teil von unterschiedlichen Workflows ist.
- Das Senden von Nachrichten an mehrere Subscriber und Antworten von Empfängern in Echtzeit nicht erforderlich sind.
- Das System oder die Anwendung kann Ereigniskonsistenz für Daten oder Zustände tolerieren.
- Die Anwendung oder Komponente muss mit anderen Anwendungen oder Services kommunizieren, die möglicherweise andere Sprachen, Protokolle oder Plattformen verwenden.

Fehler und Überlegungen

- Subscriber-Verfügbarkeit: Der Publisher weiß nicht, ob die Subscriber Listener sind, und vielleicht sind sie das auch nicht. Veröffentlichte Nachrichten sind flüchtiger Natur und können verworfen werden, wenn die Subscriber nicht verfügbar sind.
- Garantie für die Nachrichtenzustellung: In der Regel kann das Publish-Subscribe-Muster nicht die Zustellung von Nachrichten an alle Subscriber-Typen garantieren. Bestimmte Services wie Amazon Simple Notification Service (Amazon SNS) können jedoch eine exactly-once Zustellung an bestimmte Subscriber-Gruppen gewährleisten.
- Time to Live (TTL): Nachrichten haben eine Lebensdauer und laufen ab, wenn sie nicht innerhalb dieses Zeitraums verarbeitet werden. Erwägen Sie, die veröffentlichten Nachrichten einer Warteschlange hinzuzufügen, damit sie dauerhaft gespeichert werden können, und garantiert wird, dass sie auch nach Ablauf der TTL-Periode verarbeitet werden.
- Nachrichtenrelevanz: Produzenten können als Teil der Nachrichtendaten eine Zeitspanne für die Relevanz festlegen, und die Nachricht kann nach diesem Datum verworfen werden. Erwägen Sie, die Verbraucher so zu gestalten, dass sie diese Informationen prüfen, bevor Sie entscheiden, wie die Nachricht verarbeitet werden soll.
- Ereigniskonsistenz: Es gibt eine Verzögerung zwischen dem Zeitpunkt, an dem die Nachricht veröffentlicht wird, und dem Zeitpunkt, an dem sie vom Subscriber abgerufen wird. Dies kann dazu führen, dass die Subscriber-Datenspeicher mit der Zeit konsistent werden, wenn eine hohe Konsistenz erforderlich ist. Ereigniskonsistenz kann auch dann ein Problem sein, wenn Produzenten und Verbraucher nahezu in Echtzeit interagieren müssen.
- Unidirektionale Kommunikation: Das Publish-Subscribe-Muster wird als unidirektional betrachtet. Anwendungen, die einen bidirektionalen Nachrichtenaustausch mit einem Rückkanal für Abonnements benötigen, sollten ein Anfrage-Antwort-Muster verwenden, wenn eine synchrone Antwort erforderlich ist.
- Nachrichtenreihenfolge: Die Reihenfolge der Nachrichten ist nicht sichergestellt. Wenn Verbraucher sortierte Nachrichten benötigen, empfehlen wir Ihnen, Amazon-SNS-FIFO-Themen zu verwenden, um die Reihenfolge zu garantieren.
- Vervielfältigung von Nachrichten: Je nach Nachrichteninfrastruktur können doppelte Nachrichten an Verbraucher zugestellt werden. Die Verbraucher müssen so konzipiert sein, dass sie für die Verarbeitung doppelter Nachrichten idempotent sind. Verwenden Sie alternativ Amazon-SNS-FIFO-Themen, um sicherzustellen, dass die Zustellung exakt einmal erfolgt.
- Nachrichtenfilterung: Verbraucher sind oft nur an einer Teilmenge der von einem Produzenten veröffentlichten Nachrichten interessiert. Stellen Sie Mechanismen bereit, mit denen Subscriber die

Nachrichten, die sie erhalten, filtern oder eingrenzen können, indem Sie Themen oder Inhaltsfilter angeben.

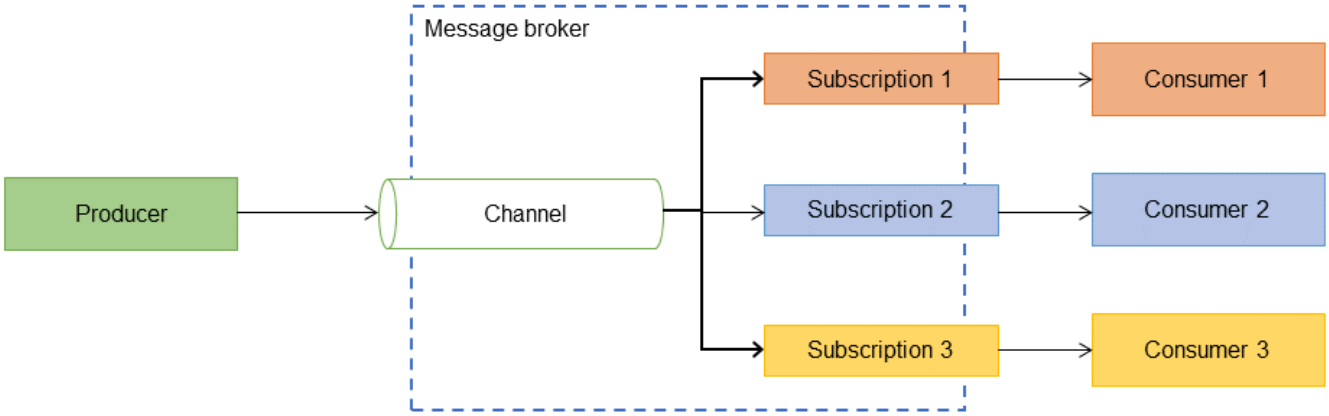
- Nachrichtenwiedergabe: Die Funktionen zur Nachrichtenwiedergabe hängen möglicherweise von der Nachrichteninfrastruktur ab. Je nach Anwendungsfall können Sie auch benutzerdefinierte Implementierungen bereitstellen.
- Warteschlangen für unzustellbare Nachrichten: In einem Postsystem ist ein Büro für unzustellbare Briefe eine Einrichtung zur Bearbeitung unzustellbarer Post. Im pub/sub-Messaging ist eine Warteschlange für unzustellbare Nachrichten eine Warteschlange für Nachrichten, die nicht an einen abonnierten Endpunkt gesendet werden können.

Implementierung

Hochrangige Architektur

In einem Publish-Subscribe-Muster verfolgt das asynchrone Nachrichten-Subsystem, auch Message Broker oder Router genannt, die Abonnements. Wenn ein Produzent ein Ereignis veröffentlicht, sendet die Nachrichten-Infrastruktur eine Nachricht an jeden Verbraucher. Nachdem eine Nachricht an Subscriber gesendet wurde, wird sie aus der Nachrichteninfrastruktur entfernt, sodass sie nicht erneut abgespielt werden kann und neue Subscriber das Ereignis nicht sehen. Message Broker oder Router entkoppeln den Ereignisproduzenten von den Nachrichtenverbrauchern, indem sie:

- Einen Eingangskanal für den Produzenten bereitstellen, um Ereignisse, die in Nachrichten verpackt sind, unter Verwendung eines definierten Nachrichtenformats zu veröffentlichen.
- Erstellung eines individuellen Ausgabekanals pro Abonnement. Ein Abonnement ist die Verbindung des Verbrauchers, über die er auf Ereignisnachrichten wartet, die einem bestimmten Eingangskanal zugeordnet sind.
- Kopieren von Nachrichten aus dem Eingabekanal in den Ausgabekanal für alle Verbraucher, wenn das Ereignis veröffentlicht wird.



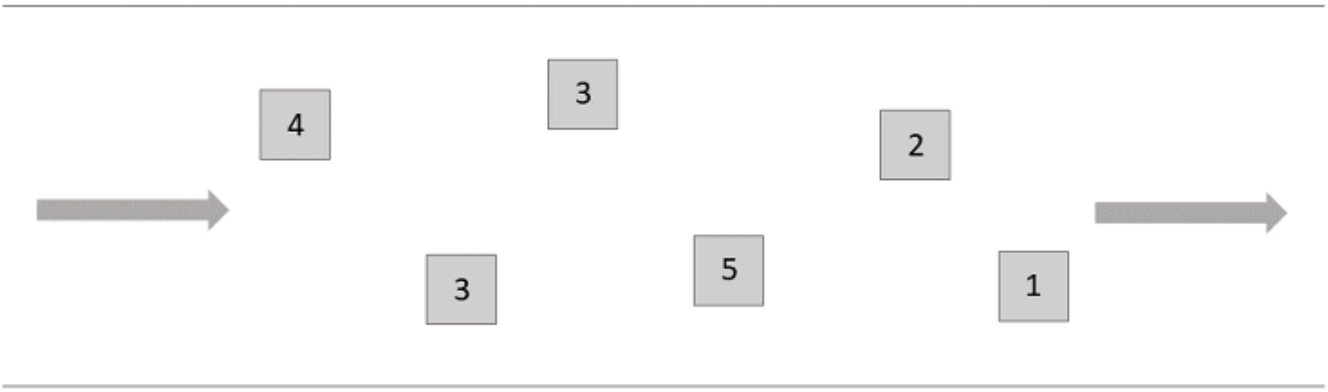
Implementierung mithilfe von AWS-Services

Amazon SNS

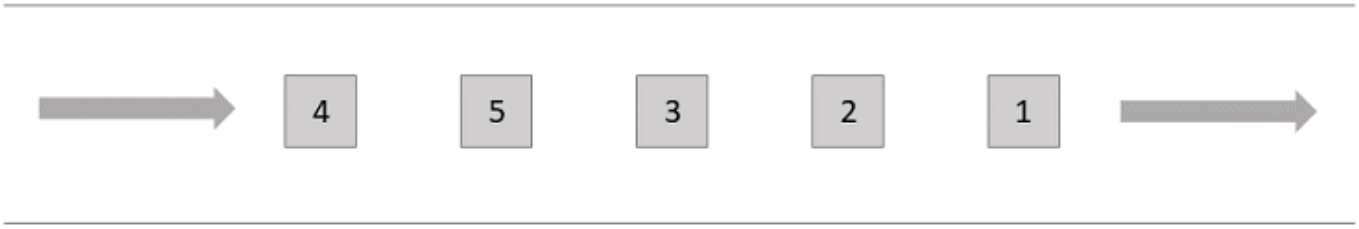
Amazon SNS ist ein vollständig verwalteter Publisher-Subscriber-Service, der Application-to-Application-Messaging (A2A) bietet, um verteilte Anwendungen zu entkoppeln. Es bietet auch Application-to-Person (A2P)-Messaging für das Senden von SMS, E-Mail und anderen Push-Benachrichtigungen.

Amazon SNS bietet zwei Arten von Themen: Standard und First-in, First-Out-(FIFO).

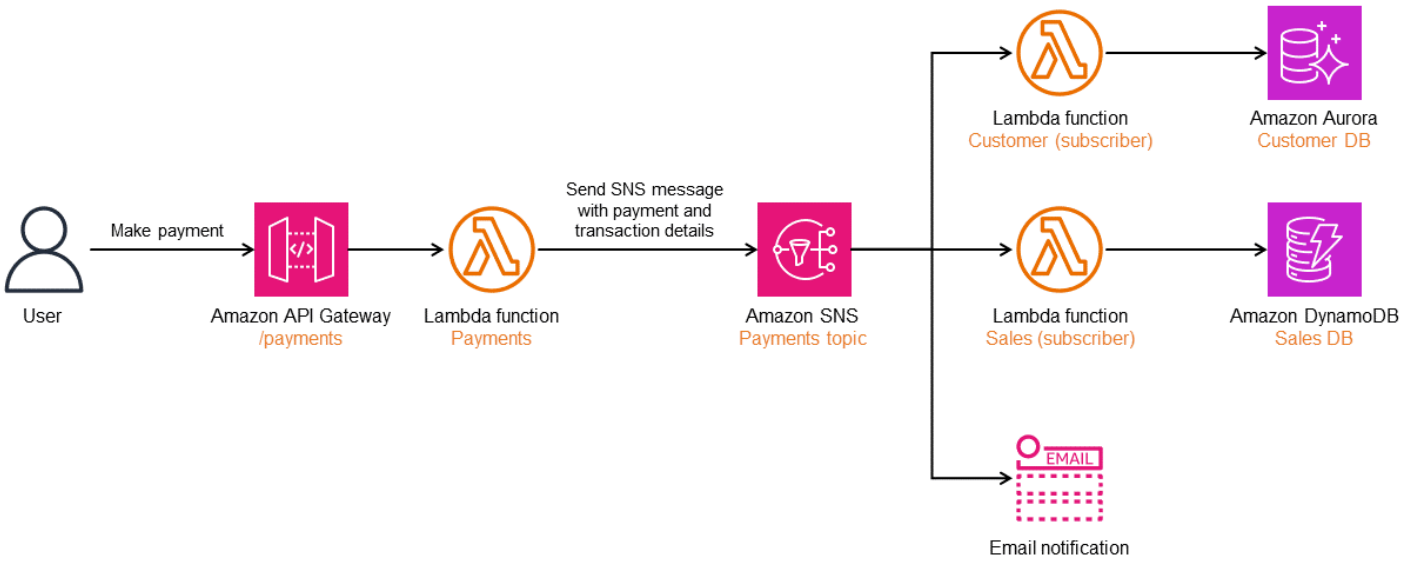
- Standardthemen unterstützen eine unbegrenzte Anzahl von Nachrichten pro Sekunde und bieten bestmögliche Sortierung und Deduplizierung.



- FIFO-Themen bieten eine strikte Reihenfolge und Deduplizierung und unterstützen bis zu 300 Nachrichten pro Sekunde oder 10 MB pro Sekunde pro FIFO-Thema (je nachdem, was zuerst eintritt).



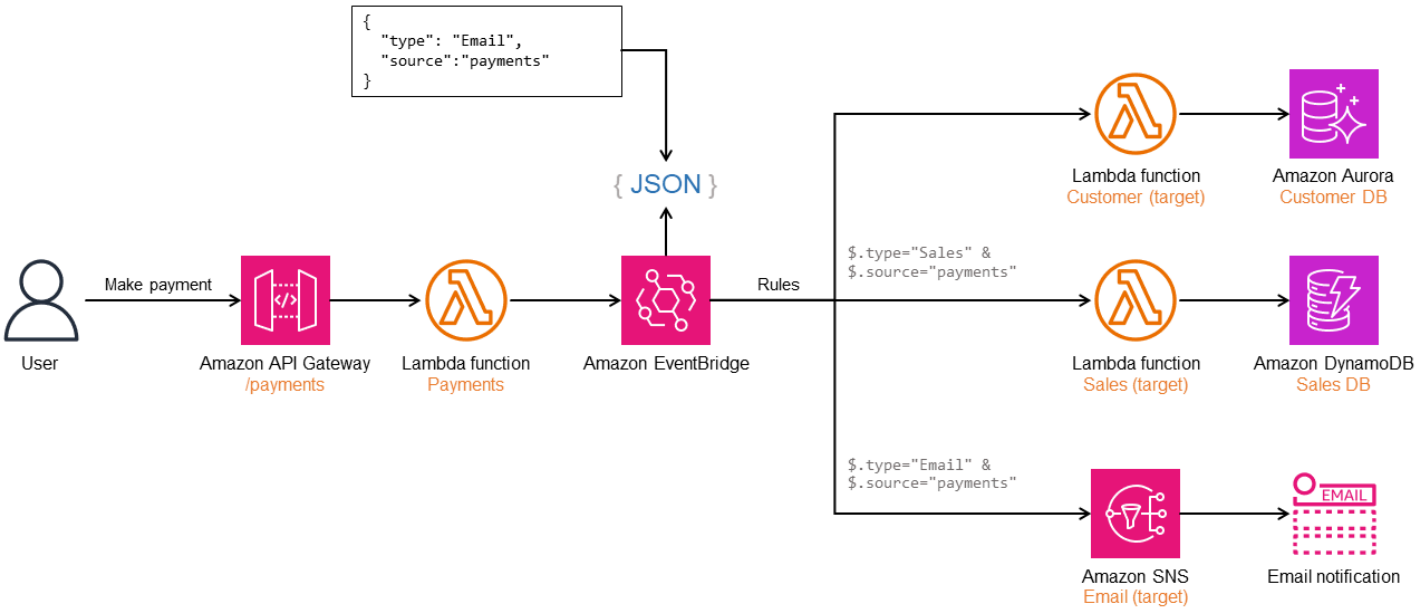
Die folgende Abbildung zeigt, wie Sie Amazon SNS verwenden können, um das Publish-Subscribe-Muster zu implementieren. Nachdem ein Benutzer eine Zahlung getätigt hat, wird von der `Payments`-Lambda-Funktion eine SNS-Nachricht an das `Payments`-SNS-Thema gesendet. Dieses SNS-Thema hat drei Subscriber. Jeder Subscriber erhält eine Kopie der Nachricht und verarbeitet sie.



Amazon EventBridge

Sie können Amazon EventBridge verwenden, wenn Sie eine komplexere Weiterleitung von Nachrichten von mehreren Produzenten über verschiedene Protokolle an abonnierte Verbraucher oder für Direkte- und Fanout-Abonnements benötigen. EventBridge unterstützt auch inhaltsbasiertes Routing, Filterung, Sequenzierung und Aufteilung oder Aggregation. In der folgenden Abbildung wird EventBridge verwendet, um eine Version des Publish-Subscribe-Musters zu erstellen, in dem Subscriber mithilfe von Ereignisregeln definiert werden. Nachdem ein Benutzer eine Zahlung getätigt hat, sendet die `Payments`-Lambda-Funktion eine Nachricht an EventBridge, indem sie den Standard-Event-Bus auf der Grundlage eines benutzerdefinierten Schemas verwendet, das drei Regeln enthält,

die auf verschiedene Ziele verweisen. Jeder Microservice verarbeitet die Nachrichten und führt die erforderlichen Aktionen aus.



Workshop

- Aufbau ereignisgesteuerter Architekturen in AWS
- Senden von Fanout-Ereignisbenachrichtigungen mit Amazon Simple Queue Service (Amazon SQS) und Amazon Simple Notification Service (Amazon SNS)

Blog-Referenzen

- Wählen zwischen Messaging-Services für Serverless-Anwendungen
- Entwicklung langlebiger Serverless-Anwendungen mit DLQs für Amazon SNS, Amazon SQS, AWS Lambda
- Vereinfachen Sie Ihre pub/sub-Nachrichten mit der Amazon-SNS-Nachrichtenfilterung

Verwandter Inhalt

- Features von pub/sub-Nachrichten

Versuchen Sie es erneut mit dem Backoff-Muster

Absicht

Das Muster „Wiederholung mit Backoff“ verbessert die Stabilität der Anwendung, indem Operationen, die aufgrund vorübergehender Fehler fehlschlagen, transparent wiederholt werden.

Motivierung

In verteilten Architekturen können vorübergehende Fehler durch Dienstdrosselung, vorübergehenden Verlust der Netzwerkkonnektivität oder vorübergehende Nichtverfügbarkeit von Diensten verursacht werden. Das automatische Wiederholen von Vorgängen, die aufgrund dieser vorübergehenden Fehler fehlschlagen, verbessert die Benutzererfahrung und die Stabilität der Anwendung. Häufige Wiederholungsversuche können jedoch die Netzwerkbandbreite überlasten und zu Konflikten führen. Exponentielles Backoff ist eine Technik, bei der Operationen wiederholt werden, indem die Wartezeiten für eine bestimmte Anzahl von Wiederholungsversuchen verlängert werden.

Anwendbarkeit

Verwenden Sie das Muster „Wiederholungsversuch mit Backoff“, wenn:

- Ihre Dienste drosseln die Anfrage häufig, um eine Überlastung zu vermeiden, was zu einem429 Zu viele AnfragenAusnahme beim Anrufvorgang.
- Das Netzwerk ist ein unsichtbarer Akteur verteilter Architekturen, und vorübergehende Netzwerkprobleme führen zu Ausfällen.
- Der aufgerufene Dienst ist vorübergehend nicht verfügbar, was zu Fehlern führt. Häufige Wiederholungsversuche können zu einer Beeinträchtigung des Dienstes führen, sofern Sie nicht mithilfe dieses Musters ein Backoff-Timeout einführen.

Probleme und Überlegungen

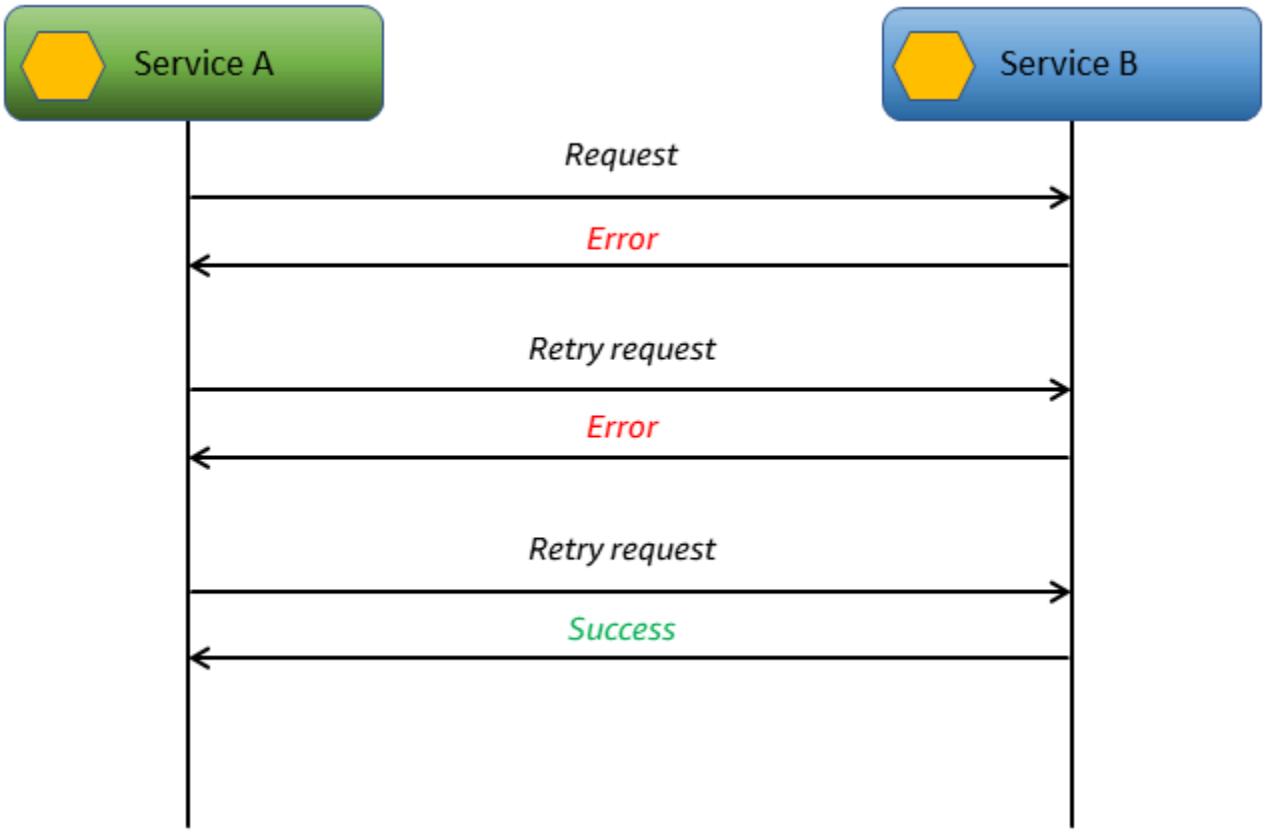
- Idempotenz: Wenn mehrere Aufrufe der Methode die gleiche Wirkung haben wie ein einziger Aufruf auf den Systemstatus, wird die Operation als idempotent angesehen. Operationen sollten idempotent sein, wenn Sie das Muster „Wiederholungsversuch mit Backoff“ verwenden. Andernfalls könnten Teilaktualisierungen den Systemstatus beschädigen.

- Netzwerk-Bandbreite: Wenn zu viele Wiederholungsversuche die Netzwerkbandbreite beanspruchen, kann es zu einer Beeinträchtigung des Dienstes kommen, was zu langsamen Reaktionszeiten führt.
- Szenarien, in denen schnell scheitern: Wenn Sie bei nicht vorübergehenden Fehlern die Ursache des Fehlers ermitteln können, ist es effizienter, schnell zu versagen, indem Sie das Schutzschaltermuster verwenden.
- Rückzahlungsrate: Die Einführung eines exponentiellen Backoffs kann sich auf das Service-Timeout auswirken und zu längeren Wartezeiten für den Endbenutzer führen.

Implementierung

Hochrangige Architektur

Das folgende Diagramm zeigt, wie Service A die Aufrufe von Service B wiederholen kann, bis eine erfolgreiche Antwort zurückgegeben wird. Wenn Service B nach einigen Versuchen keine erfolgreiche Antwort zurückgibt, kann Service A die Wiederholung beenden und dem Anrufer einen Fehler melden.

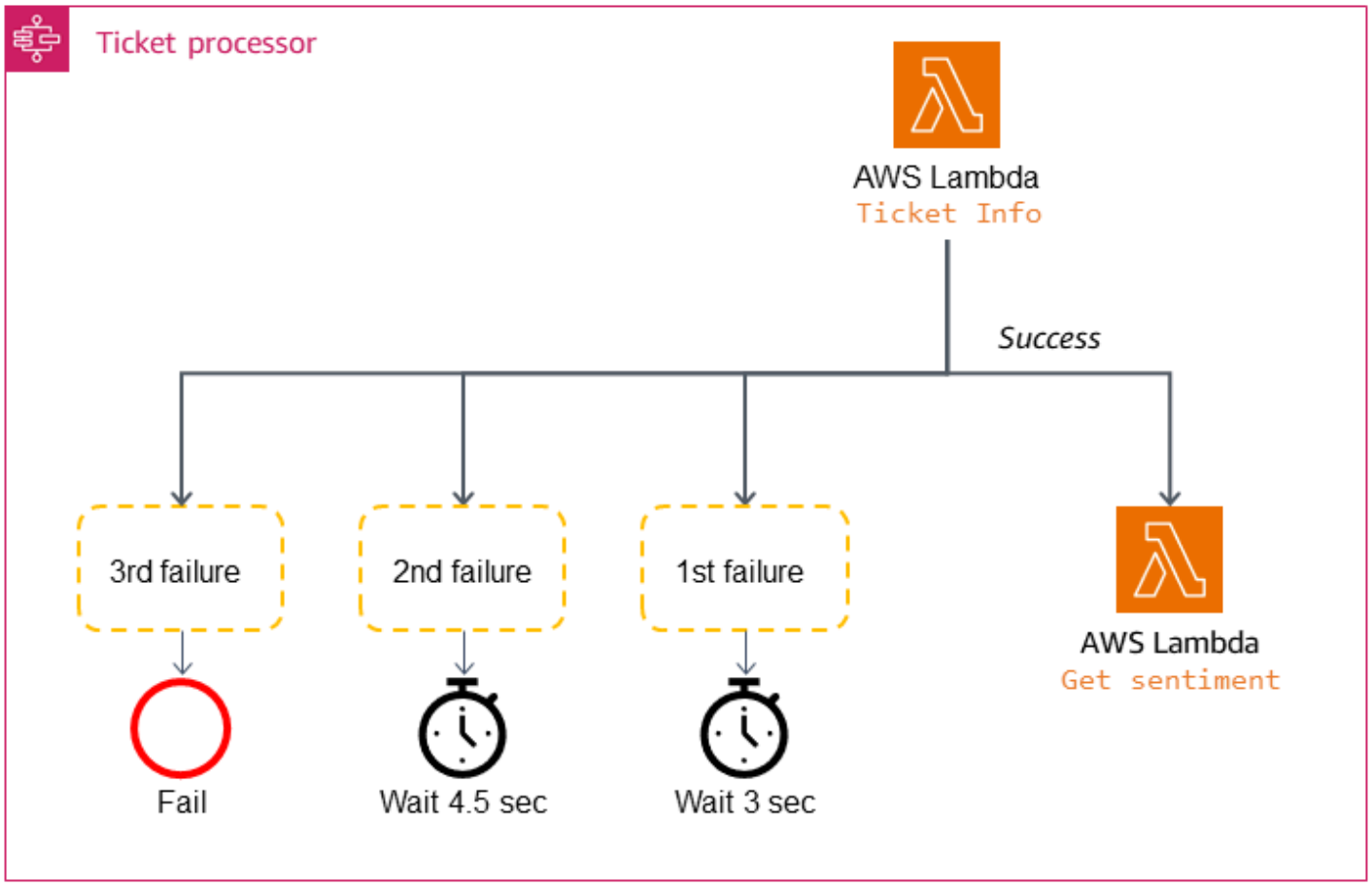


Umsetzung mitAWSDienstleistungen

Das folgende Diagramm zeigt einen Workflow zur Ticketverarbeitung auf einer Kundensupportplattform. Tickets von unzufriedenen Kunden werden beschleunigt, indem die Ticketpriorität automatisch erhöht wird. Der`Ticket info`Die Lambda-Funktion extrahiert die Ticketdetails und ruft den`Get sentiment`Lambda-Funktion. Der`Get sentiment`Die Lambda-Funktion überprüft die Meinung des Kunden, indem sie die Beschreibung anAmazon Comprehend(nicht abgebildet).

Wenn der Anruf an die`Get sentiment`Die Lambda-Funktion schlägt fehl, der Workflow versucht den Vorgang dreimal erneut.AWS Step Functionsermöglicht exponentielles Backoff, indem Sie den Backoff-Wert konfigurieren können.

In diesem Beispiel werden maximal drei Wiederholungsversuche mit einem Erhöhungsmultiplikator von 1,5 Sekunden konfiguriert. Wenn die erste Wiederholung nach 3 Sekunden erfolgt, erfolgt die zweite Wiederholung nach 3 x 1,5 Sekunden = 4,5 Sekunden, und die dritte Wiederholung erfolgt nach 4,5 x 1,5 Sekunden = 6,75 Sekunden. Wenn der dritte Wiederholungsversuch nicht erfolgreich ist, schlägt der Workflow fehl. Für die Backoff-Logik ist kein benutzerdefinierter Code erforderlich. Sie wird als Konfiguration bereitgestellt vonAWS Step Functions.



Beispiel-Code

Der folgende Code zeigt die Implementierung des Musters „Wiederholung mit Backoff“.

```
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);
  }
```

GitHubEndlager

Eine vollständige Implementierung der Beispielarchitektur für dieses Muster finden Sie in derGitHubRepositorium beihttps://github.com/aws-samples/retry-with-backoff.

Verwandter Inhalt

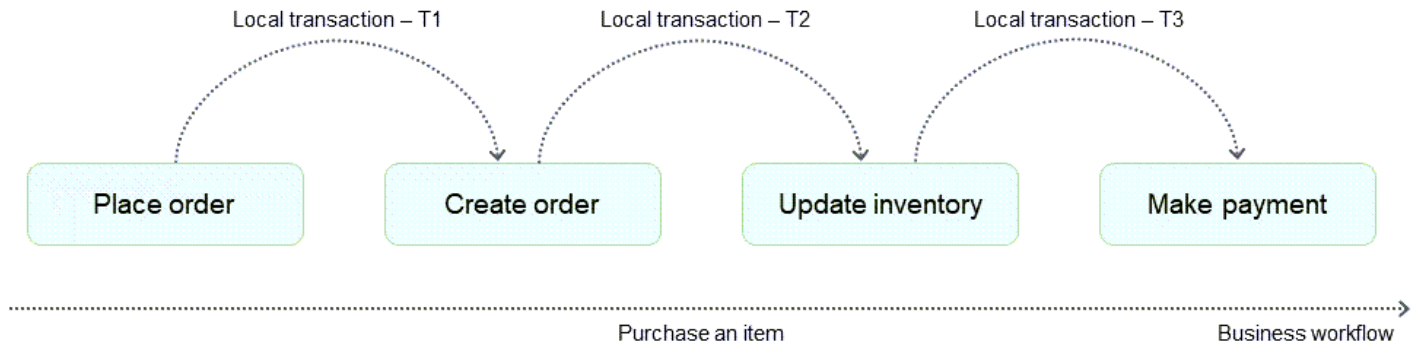
- Timeouts, Wiederholungsversuche und Backoff mit Jitter(Amazon Builders' Library)

Saga-Muster

Eine Saga besteht aus einer Abfolge von lokalen Transaktionen. Jede lokale Transaktion in einer Saga aktualisiert die Datenbank und löst die nächste lokale Transaktion aus. Wenn eine Transaktion fehlschlägt, führt die Saga kompensierende Transaktionen aus, um die von den vorherigen Transaktionen vorgenommenen Datenbankänderungen rückgängig zu machen.

Diese Abfolge von lokalen Transaktionen trägt dazu bei, einen Geschäfts-Workflow zu erreichen, indem Fortsetzungs- und Kompensationsprinzipien verwendet werden. Das Fortsetzungsprinzip entscheidet über die Vorwärtswiederherstellung des Workflows, während das Kompensationsprinzip über die Rückwärtswiederherstellung entscheidet. Wenn die Aktualisierung in irgendeinem Schritt der Transaktion fehlschlägt, veröffentlicht die Saga ein Ereignis, um entweder die Transaktion fortzusetzen (um sie erneut zu versuchen) oder zu kompensieren (um zum vorherigen Datenzustand zurückzukehren). Dadurch wird sichergestellt, dass die Datenintegrität gewahrt bleibt und in allen Datenspeichern konsistent ist.

Wenn ein Benutzer beispielsweise ein Buch bei einem Online-Händler kauft, besteht der Prozess aus einer Abfolge von Transaktionen – wie Auftragserstellung, Inventaraktualisierung, Zahlung und Versand –, die einen Geschäfts-Workflow darstellen. Um diesen Workflow abzuschließen, gibt die verteilte Architektur eine Abfolge von lokalen Transaktionen aus, um eine Bestellung in der Bestelldatenbank zu erstellen, die Inventardatenbank zu aktualisieren und die Zahlungsdatenbank zu aktualisieren. Wenn der Vorgang erfolgreich ist, werden diese Transaktionen nacheinander aufgerufen, um den Geschäfts-Workflows abzuschließen, wie das folgende Diagramm zeigt. Wenn jedoch eine dieser lokalen Transaktionen fehlschlägt, sollte das System in der Lage sein, sich für einen geeigneten nächsten Schritt zu entscheiden, d. h. entweder eine Vorwärts- oder eine Rückwärtswiederherstellung.



Anhand der folgenden beiden Szenarien lässt sich feststellen, ob der nächste Schritt eine Vorwärts- oder eine Rückwärtswiederherstellung sein sollte:

- Fehler auf Plattformebene, bei dem etwas mit der zugrunde liegenden Infrastruktur defekt ist und die Transaktion zum Fehlschlagen bringt. In diesem Fall kann das Saga-Muster eine Vorwärtswiederherstellung durchführen, indem die lokale Transaktion erneut versucht und der Geschäftsprozess fortgesetzt wird.
- Fehler auf Anwendungsebene, bei dem der Zahlungsservice aufgrund einer ungültigen Zahlung fehlschlägt. In diesem Fall kann das Saga-Muster eine Rückwärtswiederherstellung durchführen, indem es eine Ausgleichstransaktion durchführt, um das Inventar und die Bestelldatenbanken zu aktualisieren und ihren vorherigen Status wiederherzustellen.

Das Saga-Muster verwaltet den Geschäfts-Workflow und stellt sicher, dass durch die Vorwärtswiederherstellung ein gewünschter Endzustand erreicht wird. Bei Ausfällen werden die lokalen Transaktionen mithilfe der Rückwärtswiederherstellung rückgängig gemacht, um Probleme mit der Datenkonsistenz zu vermeiden.

Das Saga-Muster hat zwei Varianten: Choreographie und Orchestrierung.

Saga-Choreographie

Das Saga-Choreographiemuster hängt von den Ereignissen ab, die von den Microservices veröffentlicht werden. Die Saga-Teilnehmer (Microservices) abonnieren die Ereignisse und handeln auf der Grundlage der Ereignisauslöser. Beispielsweise gibt der Bestellservice im folgenden Diagramm ein `OrderPlaced`-Ereignis aus. Der Inventarservice abonniert dieses Ereignis und aktualisiert das Inventar, wenn das `OrderPlaced`-Ereignis ausgegeben wird. In ähnlicher Weise verhalten sich die teilnehmenden Services auf Grundlage des Kontextes des ausgegebenen Ereignisses.

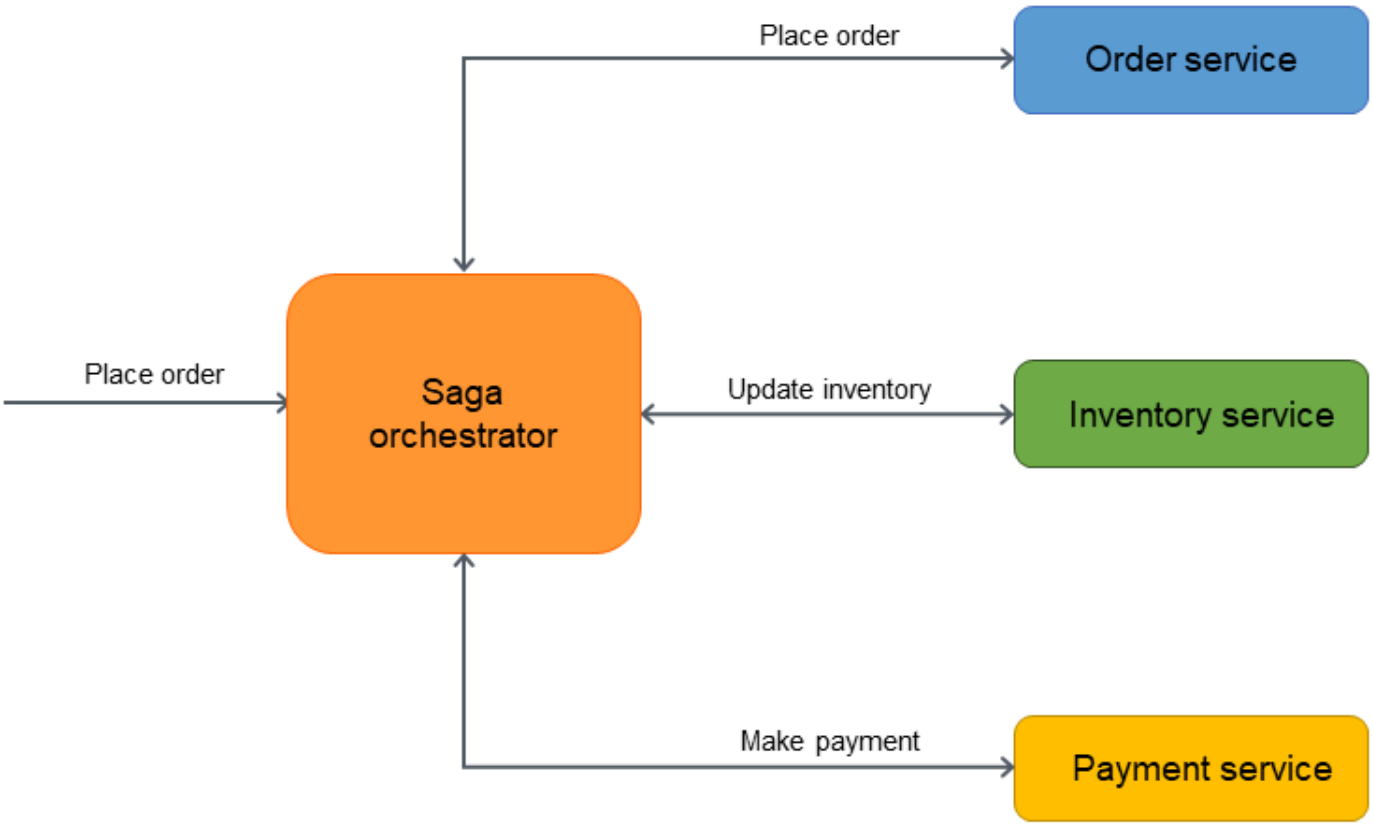
Das Saga-Choreographiemuster eignet sich, wenn nur wenige Teilnehmer an der Saga beteiligt sind und Sie eine einfache Implementierung benötigen, bei der es keine einzelne Fehlerquelle gibt. Wenn mehr Teilnehmer hinzugefügt werden, wird es schwieriger, die Abhängigkeiten zwischen den Teilnehmern anhand dieses Musters nachzuverfolgen.

Eine ausführliche Übersicht finden Sie im Abschnitt zur Saga-Choreographie dieses Handbuchs.

Saga-Orchestrierung

Das Saga-Orchestrierungsmuster hat einen zentralen Koordinator, der als Orchestrator bezeichnet wird. Der Saga-Orchestrator verwaltet und koordiniert den gesamten Transaktionslebenszyklus. Er ist sich der Reihe von Schritten bewusst, die ausgeführt werden müssen, um die Transaktion abzuschließen. Um einen Schritt auszuführen, sendet er eine Nachricht an den Teilnehmer-Microservice, der den Vorgang ausführen soll. Der Teilnehmer-Microservice schließt den Vorgang ab und sendet eine Nachricht zurück an den Orchestrator. Auf der Grundlage der empfangenen Nachricht entscheidet der Orchestrator, welcher Microservice als Nächstes in der Transaktion ausgeführt werden soll.

Das Saga-Orchestrierungs-Muster eignet sich, wenn es viele Teilnehmer gibt und eine lose Verkoppelung zwischen den Saga-Teilnehmern erforderlich ist. Der Orchestrator kapselt die Komplexität in der Logik ab, indem er die lose Verkoppelung der Teilnehmer sicherstellt. Der Orchestrator kann jedoch zu einem einzelnen Ausfallpunkt werden, da er den gesamten Workflow steuert.



Eine ausführliche Übersicht finden Sie im Abschnitt zur Saga-Orchestrierung dieses Handbuchs.

Saga-Choreographie-Muster

Absicht

Das Saga-Choreographiem-Mster trägt dazu bei, die Datenintegrität bei verteilten Transaktionen, die sich über mehrere Services erstrecken, mithilfe von Ereignisabonnements aufrechtzuerhalten. Bei einer verteilten Transaktion können mehrere Services aufgerufen werden, bevor eine Transaktion abgeschlossen ist. Wenn die Services Daten in verschiedenen Datenspeichern speichern, kann es schwierig sein, die Datenkonsistenz zwischen diesen Datenspeichern zu wahren.

Motivation

Eine Transaktion ist eine einzelne Arbeitseinheit, die mehrere Schritte umfassen kann, wobei alle Schritte vollständig ausgeführt werden oder kein Schritt ausgeführt wird, was zu einem Datenspeicher führt, der seinen konsistenten Zustand beibehält. Die Begriffe Atomarität, Konsistenz, Isolation und Haltbarkeit (ACID) definieren die Eigenschaften einer Transaktion. Relationale Datenbanken bieten ACID-Transaktionen zur Wahrung der Datenkonsistenz.

Um die Konsistenz einer Transaktion aufrechtzuerhalten, verwenden relationale Datenbanken die zweiphasige Commit-Methode (2PC). Diese besteht aus einer Vorbereitungsphase und einer Commit-Phase.

- In der Vorbereitungsphase fordert der Koordinierungsprozess die an der Transaktion beteiligten Prozesse (Teilnehmer) auf, zu versprechen, die Transaktion entweder zu bestätigen oder rückgängig zu machen.
- In der Commit-Phase fordert der Koordinierungsprozess die Teilnehmer auf, die Transaktion zu bestätigen. Wenn sich die Teilnehmer in der Vorbereitungsphase nicht auf eine Bestätigung einigen können, wird die Transaktion zurückgesetzt.

In verteilten Systemen, die einem database-per-service Entwurfsmuster folgen, ist das zweiphasige Commit keine Option. Das liegt daran, dass jede Transaktion auf verschiedene Datenbanken verteilt ist und es keinen einzelnen Controller gibt, der einen Prozess koordinieren kann, der dem zweiphasigen Commit in relationalen Datenspeichern ähnelt. In diesem Fall besteht eine Lösung darin, das Saga-Choreographie-Muster zu verwenden.

Anwendbarkeit

Verwenden Sie das Saga-Choreographie-Muster, wenn:

- Ihr System Datenintegrität und Konsistenz bei verteilten Transaktionen erfordert, die sich über mehrere Datenspeicher erstrecken.
- Der Datenspeicher (z. B. eine NoSQL-Datenbank) bietet kein 2PC, um ACID-Transaktionen zu ermöglichen. Sie müssen mehrere Tabellen innerhalb einer einzigen Transaktion aktualisieren, und die Implementierung von 2PC innerhalb der Anwendungsgrenzen wäre eine komplexe Aufgabe.
- Ein zentraler Steuerungsprozess, der die Transaktionen der Teilnehmer verwaltet, könnte zu einem einzelnen Ausfallpunkt werden.
- Bei Saga-Teilnehmern handelt es sich um unabhängige Services, die eine lose Verkoppelung aufweisen müssen.
- In einer Geschäftsdomain findet Kommunikation zwischen begrenzten Kontexten statt.

Fehler und Überlegungen

- Komplexität: Da die Anzahl der Microservices zunimmt, kann es aufgrund der Anzahl der Interaktionen zwischen den Microservices schwierig werden, Saga-Choreographie zu verwalten. Darüber hinaus erhöhen Ausgleichstransaktionen und Wiederholungsversuche die Komplexität des Anwendungscodes, was zu einem Wartungsaufwand führen kann. Choreographie eignet sich, wenn nur wenige Teilnehmer an der Saga beteiligt sind und Sie eine einfache Implementierung benötigen, bei der es keinen einzelnen Ausfallpunkt gibt. Wenn mehr Teilnehmer hinzugefügt werden, wird es schwieriger, die Abhängigkeiten zwischen den Teilnehmern anhand dieses Musters nachzuverfolgen.
- Ausfallsichere Implementierung: In der Saga-Choreographie ist es schwieriger, Timeouts, Wiederholungen und andere Resilienzmuster global zu implementieren als bei der Saga-Orchestrierung. Choreographie muss auf einzelnen Komponenten statt auf Orchestrator-Ebene implementiert werden.
- Zyklische Abhängigkeiten: Die Teilnehmer konsumieren Nachrichten, die voneinander veröffentlicht werden. Dies kann zu zyklischen Abhängigkeiten führen, was zu Codekomplexität und Wartungsaufwand sowie zu möglichen Blockierungen führen kann.
- Problem mit dualen Schreibvorgängen: Der Microservice muss die Datenbank atomar aktualisieren und ein Ereignis veröffentlichen. Wenn einer der beiden Vorgänge fehlschlägt, kann dies zu einem inkonsistenten Zustand führen. Eine Möglichkeit, dieses Problem zu lösen, besteht darin, das Transactional-Outbox-Muster zu verwenden.
- Bewahrung von Ereignissen: Die Saga-Teilnehmer handeln auf Grundlage der veröffentlichten Ereignisse. Für Prüf-, Debugging- und Wiedergabezwecke ist es wichtig, die Ereignisse in

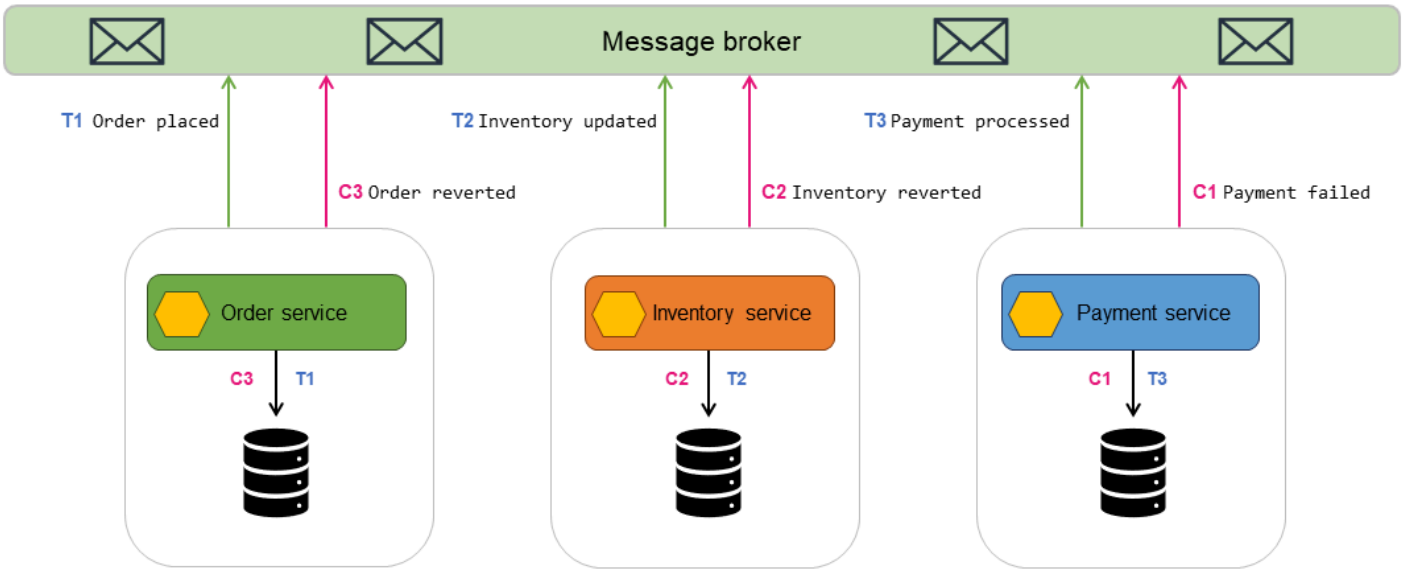
der Reihenfolge zu speichern, in der sie auftreten. Sie können das Ereignis-Sourcing-Muster verwenden, um die Ereignisse in einem Ereignisspeicher zu speichern, falls zur Wiederherstellung der Datenkonsistenz eine Wiedergabe des Systemstatus erforderlich ist. Ereignisspeicher können auch für Prüfungs- und Problembehandlungszwecke verwendet werden, da sie jede Änderung im System widerspiegeln.

- Ereigniskonsistenz: Die sequentielle Verarbeitung lokaler Transaktionen führt zur Ereigniskonsistenz, was bei Systemen, die eine hohe Konsistenz erfordern, eine Herausforderung darstellen kann. Sie können dieses Problem lösen, indem Sie die Erwartungen Ihrer Geschäftsteams an das Konsistenzmodell festlegen oder auf einen Datenspeicher umsteigen, der eine hohe Konsistenz bietet.
- Idempotenz: Saga-Teilnehmer müssen idempotent sein, um bei vorübergehenden Ausfällen, die durch unerwartete Abstürze und Orchestrator-Ausfälle verursacht werden, eine wiederholte Ausführung zu ermöglichen.
- Transaktionsisolierung: Dem Saga-Muster fehlt die Transaktionsisolierung, die eine der vier Eigenschaften von ACID-Transaktionen ist. Der Grad der Isolierung einer Transaktion bestimmt, inwieweit sich andere gleichzeitige Transaktionen auf die Daten auswirken können, mit denen die Transaktion arbeitet. Die gleichzeitige Orchestrierung von Transaktionen kann zu veralteten Daten führen. Wir empfehlen, semantische Sperren zu verwenden, um solche Szenarien zu handhaben.
- Beobachtbarkeit: Beobachtbarkeit bezieht sich auf eine detaillierte Protokollierung und Rückverfolgung, um Probleme im Implementierungs- und Orchestrierungsprozess zu beheben. Dies wird wichtig, wenn die Anzahl der Saga-Teilnehmer zunimmt, was zu einer Komplexität beim Debuggen führt. In der Saga-Choreographie sind nd-to-end E-Monitoring und Reporting schwieriger zu erreichen als bei der Orchestrierung von Saga.
- Latenzprobleme: Ausgleichstransaktionen können die Latenz der Gesamtantwortzeit erhöhen, wenn Saga aus mehreren Schritten besteht. Wenn die Transaktionen synchrone Aufrufe tätigen, kann dies die Latenz weiter erhöhen.

Implementierung

Hochrangige Architektur

Im folgenden Architekturdiagramm hat die Saga-Choreographie drei Teilnehmer: den Bestellservice, den Inventarservice und den Zahlungsservice. Drei Schritte sind erforderlich, um die Transaktion abzuschließen: T1, T2 und T3. Drei Ausgleichstransaktionen stellen den Ausgangszustand der Daten wieder her: C1, C2 und C3.



- Der Bestellservice führt eine lokale Transaktion, T1, aus, die die Datenbank atomar aktualisiert und eine `Order placed`-Nachricht an den Message Broker veröffentlicht.
- Der Inventarservice abonniert die Bestellservice-Nachrichten und empfängt die Nachricht, dass eine Bestellung erstellt wurde.
- Der Inventarservice führt eine lokale Transaktion, T2, aus, die die Datenbank atomar aktualisiert und eine `Inventory updated`-Nachricht an den Message Broker veröffentlicht.
- Der Zahlungsservice abonniert die Nachrichten vom Inventarservice und erhält die Nachricht, dass das Inventar aktualisiert wurde.
- Der Zahlungsservice führt eine lokale Transaktion, T3, durch, die die Datenbank automatisch mit Zahlungsdetails aktualisiert und eine `Payment processed`-Nachricht an den Message Broker veröffentlicht.
- Schlägt die Zahlung fehl, führt der Zahlungsservice eine Ausgleichstransaktion (C1) durch, die die Zahlung in der Datenbank automatisch rückgängig macht und eine `Payment failed`-Nachricht an den Message Broker veröffentlicht.
- Die Ausgleichstransaktionen C2 und C3 werden ausgeführt, um die Datenkonsistenz wiederherzustellen.

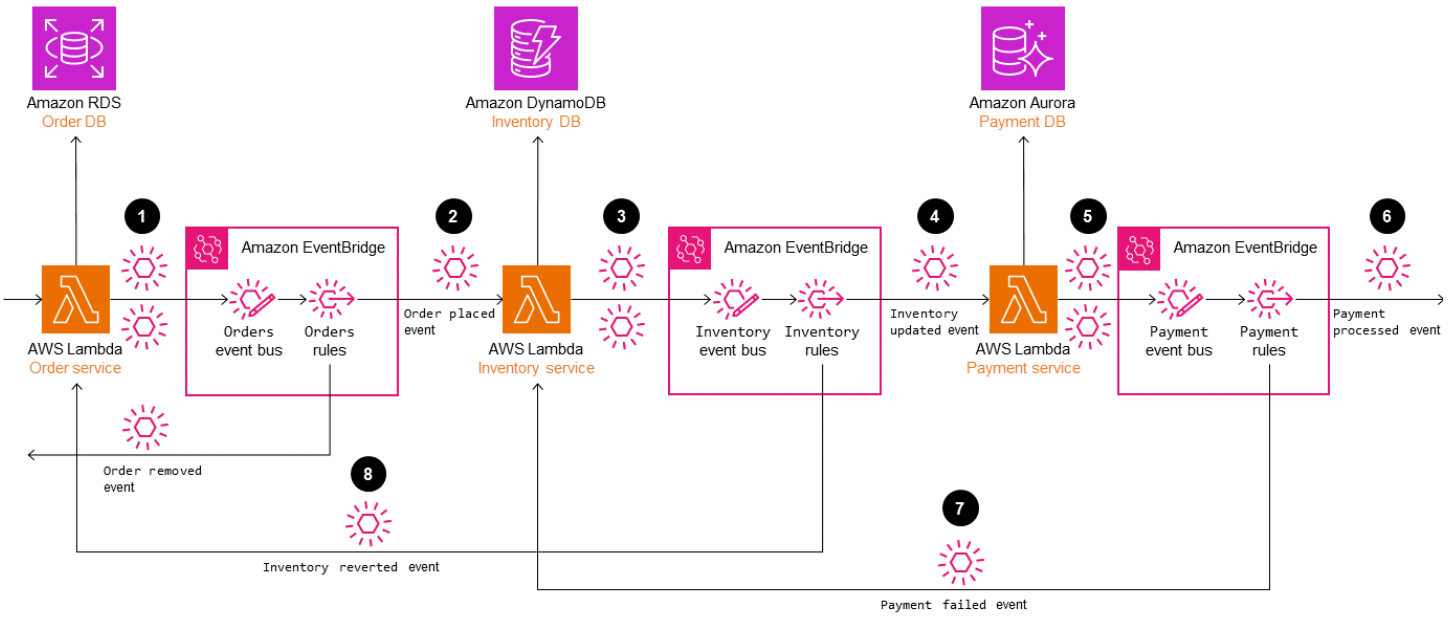
Implementierung mithilfe von AWS-Services

Sie können das Saga-Choreographie-Muster mithilfe von Amazon EventBridge implementieren. EventBridgeverwendet Ereignisse, um Anwendungskomponenten zu verbinden. Der Service

verarbeitet Ereignisse über Event Busse oder Pipes. Ein Event Bus ist ein Router, der Ereignisse empfängt und sie an null oder mehr Ziele weiterleitet. Mit dem Event Bus verknüpfte Regeln werten Ereignisse aus, sobald sie eintreffen, und senden sie zur Verarbeitung an Ziele.

In der folgenden Architektur:

- Die Microservices – Bestellservice, Inventarservice und Zahlungsservice – werden als Lambda-Funktionen implementiert.
- Es gibt drei benutzerdefinierte EventBridge Busse: `Orders` Event Bus, `Inventory` Event Bus und `Payment` Event Bus.
- `Orders`-Regeln, `Inventory`-Regeln und `Payment`-Regeln stimmen mit den Ereignissen überein, die an den entsprechenden Event Bus gesendet werden, und rufen die Lambda-Funktionen auf.



In einem erfolgreichen Szenario, wenn eine Bestellung aufgegeben wird:

1. Der Bestellservice bearbeitet die Anfrage und sendet das Ereignis an den `Orders`-Event-Bus.
2. Die `Orders`-Regeln stimmen mit den Ereignissen überein und starten den Inventarservice.
3. Der Inventarservice aktualisiert das Inventar und sendet das Ereignis an den `Inventory`-Event-Bus.
4. Die `Inventory`-Regeln stimmen mit den Ereignissen überein und starten den Zahlungsservice.

5. Der Zahlungsservice verarbeitet die Zahlung und sendet das Ereignis an den `Payment`-Event-Bus.
6. Die `Payment`-Regeln stimmen mit den Ereignissen überein und senden die `Payment processed`-Ereignisbenachrichtigung an den Listener.

   Wenn bei der Auftragsverarbeitung ein Problem auftritt, starten die EventBridge Regeln alternativ die Ausgleichstransaktionen, um die Datenaktualisierungen rückgängig zu machen, um die Datenkonsistenz und -integrität zu wahren.
7. Schlägt die Zahlung fehl, verarbeiten die `Payment`-Regeln das Ereignis und starten den Inventarservice. Der Inventarservice führt Ausgleichstransaktionen durch, um das Inventar wiederherzustellen.
8. Wenn das Inventar wiederhergestellt wurde, sendet der Inventarservice das `Inventory reverted`-Ereignis an den `Inventory`-Event-Bus. Dieses Ereignis wird durch `Inventory`-Regeln verarbeitet. Es startet den Bestellservice, der die Ausgleichstransaktion durchführt, um die Bestellung zu entfernen.

Verwandter Inhalt

- Saga-Orchestrierungs-Muster
- Transactional-Outbox-Muster
- Wiederholungsversuch mit Backoff-Muster

Saga-Orchestrierungs-Muster

Absicht

Das Saga-Orchestrierungs-Muster verwendet einen zentralen Koordinator (Orchestrator), um die Datenintegrität bei verteilten Transaktionen, die sich über mehrere Services erstrecken, aufrechtzuerhalten. Bei einer verteilten Transaktion können mehrere Services aufgerufen werden, bevor eine Transaktion abgeschlossen ist. Wenn die Services Daten in verschiedenen Datenspeichern speichern, kann es schwierig sein, die Datenkonsistenz zwischen diesen Datenspeichern zu wahren.

Motivation

Eine Transaktion ist eine einzelne Arbeitseinheit, die mehrere Schritte umfassen kann, wobei alle Schritte vollständig ausgeführt werden oder kein Schritt ausgeführt wird, was zu einem Datenspeicher

führt, der seinen konsistenten Zustand beibehält. Die Begriffe Atomarität, Konsistenz, Isolation und Haltbarkeit (ACID) definieren die Eigenschaften einer Transaktion. Relationale Datenbanken bieten ACID-Transaktionen zur Wahrung der Datenkonsistenz.

Um die Konsistenz einer Transaktion aufrechtzuerhalten, verwenden relationale Datenbanken die zweiphasige Commit-Methode (2PC). Diese besteht aus einer Vorbereitungsphase und einer Commit-Phase.

- In der Vorbereitungsphase fordert der Koordinierungsprozess die an der Transaktion beteiligten Prozesse (Teilnehmer) auf, zu versprechen, die Transaktion entweder zu bestätigen oder rückgängig zu machen.
- In der Commit-Phase fordert der Koordinierungsprozess die Teilnehmer auf, die Transaktion zu bestätigen. Wenn sich die Teilnehmer in der Vorbereitungsphase nicht auf eine Bestätigung einigen können, wird die Transaktion zurückgesetzt.

In verteilten Systemen, die einem database-per-service Entwurfsmuster folgen, ist das zweiphasige Commit keine Option. Das liegt daran, dass jede Transaktion auf verschiedene Datenbanken verteilt ist und es keinen einzelnen Controller gibt, der einen Prozess koordinieren kann, der dem zweiphasigen Commit in relationalen Datenspeichern ähnelt. In diesem Fall besteht eine Lösung darin, das Saga-Orchestrierungs-Muster zu verwenden.

Anwendbarkeit

Verwenden Sie das Saga-Orchestrierungs-Muster, wenn:

- Ihr System Datenintegrität und Konsistenz bei verteilten Transaktionen erfordert, die sich über mehrere Datenspeicher erstrecken.
- Der Datenspeicher nicht über 2PC verfügt, um ACID-Transaktionen zu ermöglichen, und die Implementierung von 2PC innerhalb der Anwendungsgrenzen ist eine komplexe Aufgabe.
- Sie haben NoSQL-Datenbanken, die keine ACID-Transaktionen bereitstellen, und Sie müssen mehrere Tabellen innerhalb einer einzigen Transaktion aktualisieren.

Fehler und Überlegungen

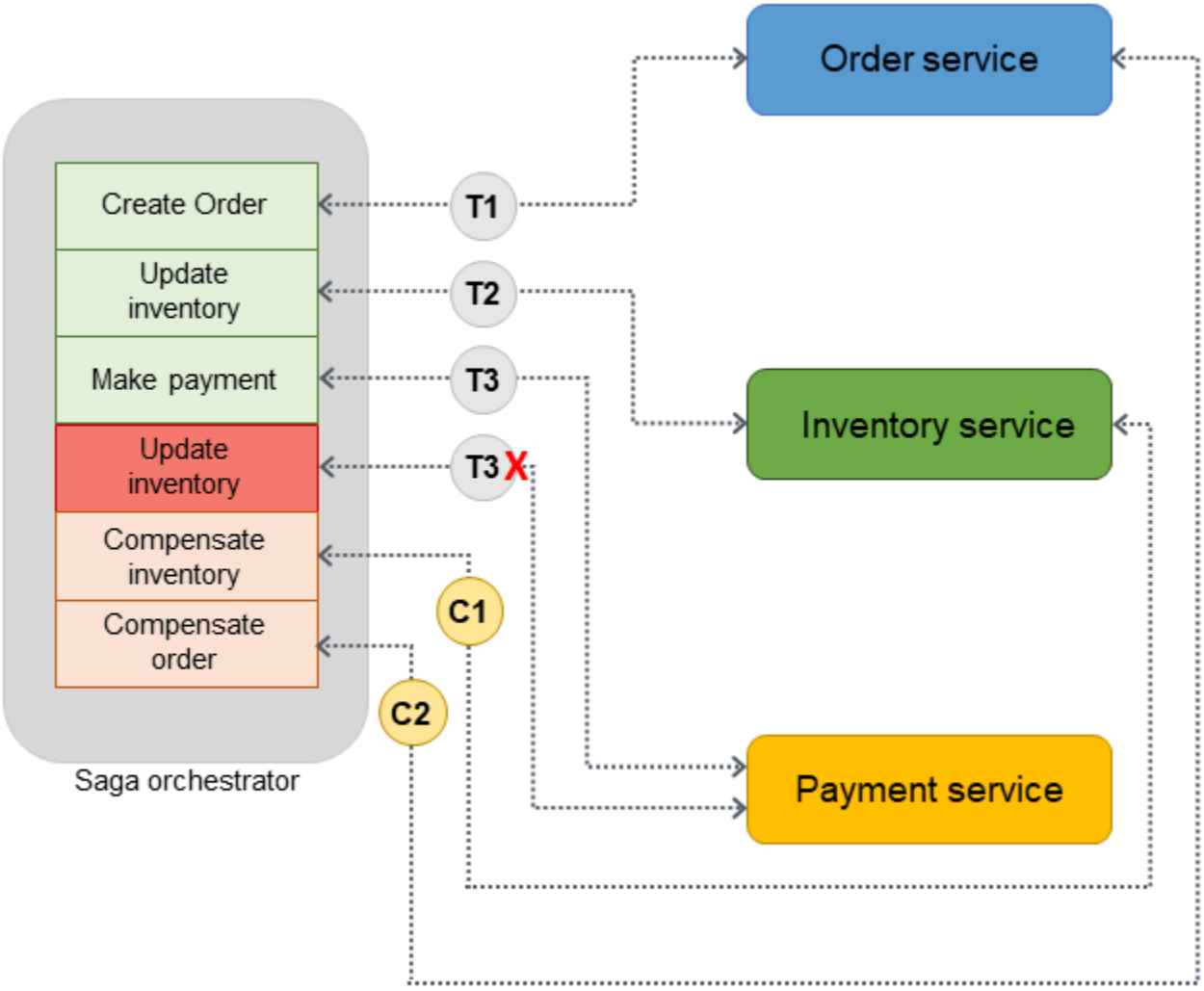
- Komplexität: Ausgleichstransaktionen und Wiederholungsversuche erhöhen die Komplexität des Anwendungscodes, was zu einem Wartungsaufwand führen kann.

- Ereigniskonsistenz: Die sequentielle Verarbeitung lokaler Transaktionen führt zur Ereigniskonsistenz, was bei Systemen, die eine hohe Konsistenz erfordern, eine Herausforderung darstellen kann. Sie können dieses Problem lösen, indem Sie die Erwartungen Ihrer Geschäftsteams an das Konsistenzmodell festlegen oder auf einen Datenspeicher umsteigen, der eine hohe Konsistenz bietet.
- Idempotenz: Saga-Teilnehmer müssen idempotent sein, um bei vorübergehenden Ausfällen, die durch unerwartete Abstürze und Orchestrator-Ausfälle verursacht werden, eine wiederholte Ausführung zu ermöglichen.
- Transaktionsisolierung: In Saga fehlt es an Transaktionsisolierung. Die gleichzeitige Orchestrierung von Transaktionen kann zu veralteten Daten führen. Wir empfehlen, semantische Sperren zu verwenden, um solche Szenarien zu handhaben.
- Beobachtbarkeit: Beobachtbarkeit bezieht sich auf eine detaillierte Protokollierung und Rückverfolgung, um Probleme im Ausführungs- und Orchestrierungsprozess zu beheben. Dies wird wichtig, wenn die Anzahl der Saga-Teilnehmer zunimmt, was zu einer Komplexität beim Debuggen führt.
- Latenzprobleme: Ausgleichstransaktionen können die Latenz der Gesamtantwortzeit erhöhen, wenn Saga aus mehreren Schritten besteht. Vermeiden Sie in solchen Fällen synchrone Aufrufe.
- Einzelner Ausfallpunkt: Der Orchestrator kann zu einem einzelnen Ausfallpunkt werden, da er die gesamte Transaktion koordiniert. In einigen Fällen wird aufgrund dieses Problems das Saga-Choreographie-Muster bevorzugt.

Implementierung

Hochrangige Architektur

Im folgenden Architekturdiagramm hat der Saga-Orchestrator drei Teilnehmer: den Bestellservice, den Inventarservice und den Zahlungsservice. Drei Schritte sind erforderlich, um die Transaktion abzuschließen: T1, T2 und T3. Der Saga-Orchestrator kennt die Schritte und führt sie in der erforderlichen Reihenfolge aus. Wenn Schritt T3 fehlschlägt (Zahlungsausfall), führt der Orchestrator die Ausgleichstransaktionen C1 und C2 aus, um den ursprünglichen Zustand der Daten wiederherzustellen.



Sie können AWS Step Functions verwenden, um die Saga-Orchestrierung zu implementieren, wenn die Transaktion auf mehrere Datenbanken verteilt ist.

Implementierung mithilfe von AWS -Services

Die Beispiellösung verwendet den Standardworkflow in Step Functions, um das Saga-Orchestrierungs-Muster zu implementieren.

Wenn ein Kunde die API aufruft, wird die Lambda-Funktion aufgerufen, und die Vorverarbeitung erfolgt in der Lambda-Funktion. Die Funktion startet den Step-Functions-Workflow, um mit der Verarbeitung der verteilten Transaktion zu beginnen. Wenn keine Vorverarbeitung erforderlich ist, können Sie den Step-Functions-Workflow direkt vom API Gateway aus starten, ohne die Lambda-Funktion zu verwenden.

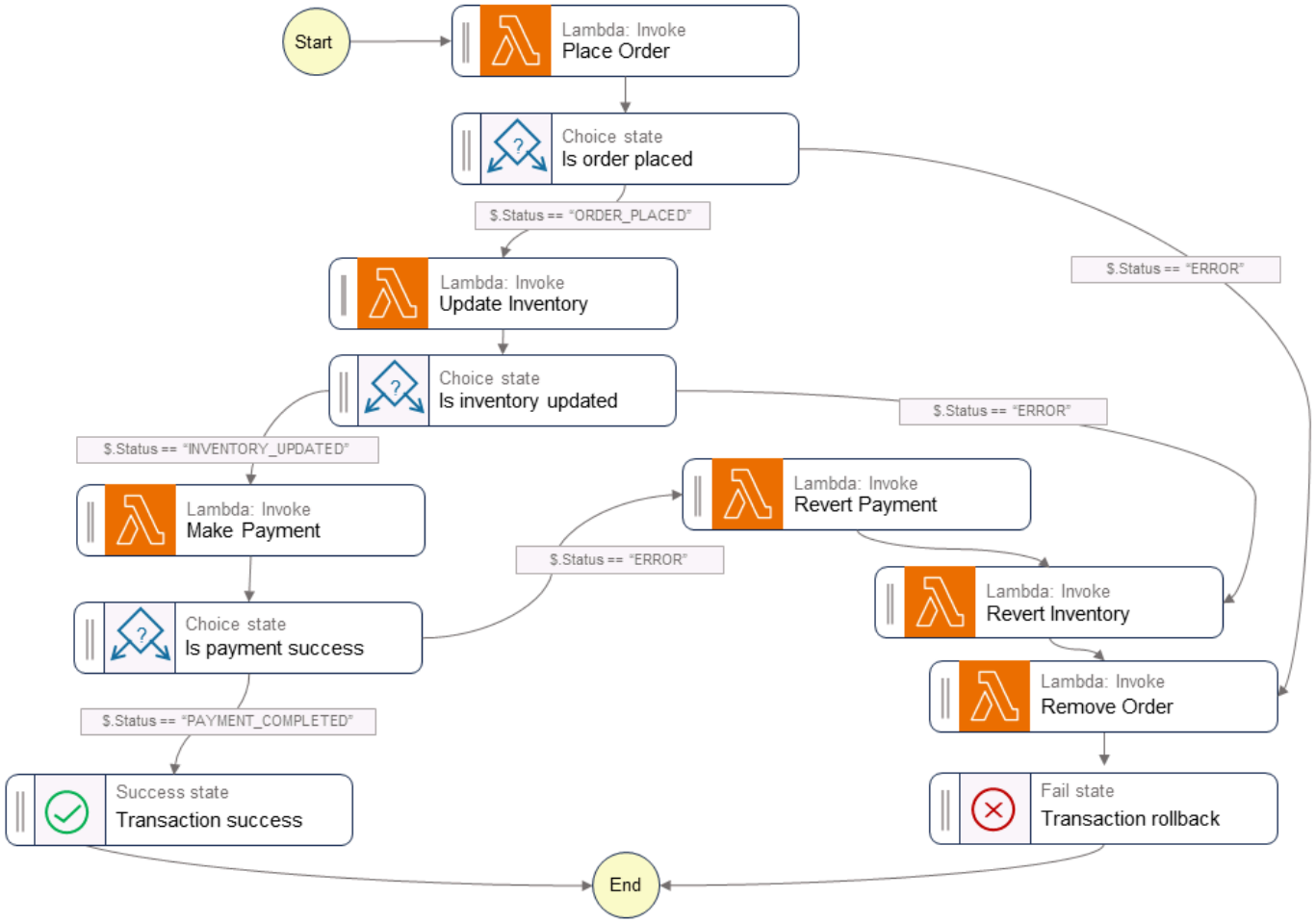
Die Verwendung von Step Functions entschärft das Problem des einzelnen Ausfallpunkts, das mit der Implementierung des Saga-Orchestrierungs-Musters verbunden ist. Step Functions verfügt über eine integrierte Fehlertoleranz und hält die Servicekapazität über mehrere Availability Zones in jeder AWS-Region aufrecht, um Anwendungen vor Ausfällen einzelner Rechner oder Rechenzentren zu schützen. Dadurch wird eine hohe Verfügbarkeit sowohl für den Service selbst als auch für den von ihm betriebenen Anwendungsworkflow gewährleistet.

Der Step-Functions-Workflow

Mit der Zustandsmaschine für Step Functions können Sie die entscheidungsbasierten Kontrollfluss-Anforderungen für die Musterimplementierung konfigurieren. Der Step-Functions-Workflow ruft die einzelnen Services für die Auftragserteilung, die Bestandsaktualisierung und die Zahlungsabwicklung auf, um die Transaktion abzuschließen, und sendet eine Ereignisbenachrichtigung zur weiteren Bearbeitung. Der Step-Functions-Workflow fungiert als Orchestrator für die Koordination der Transaktionen. Wenn der Workflow Fehler enthält, führt der Orchestrator die Ausgleichstransaktionen aus, um sicherzustellen, dass die Datenintegrität über alle Services hinweg gewahrt bleibt.

Das folgende Diagramm zeigt die Schritte, die im Step-Functions-Workflow ausgeführt werden. Die Schritte `Place Order`, `Update Inventory` und `Make Payment` geben den erfolgreichen Pfad an. Die Bestellung wird aufgegeben, das Inventar wird aktualisiert und die Zahlung wird bearbeitet, bevor dem Aufrufer ein `Success`-Status zurückgegeben wird.

Die Lambda-Funktionen `Revert Payment`, `Revert Inventory` und `Remove Order` geben die Ausgleichstransaktionen an, die der Orchestrator ausführt, wenn ein Schritt im Workflow fehlschlägt. Wenn der Workflow beim Schritt `Update Inventory` fehlschlägt, ruft der Orchestrator die Schritte `Revert Inventory` und `Remove Order` auf, bevor er dem Aufrufer einen `Fail`-Status zurückgibt. Diese Ausgleichstransaktionen stellen sicher, dass die Datenintegrität gewahrt bleibt. Das Inventar kehrt auf seinen ursprünglichen Stand zurück und die Reihenfolge wird rückgängig gemacht.



Beispiel-Code

Der folgende Beispielcode zeigt, wie Sie mithilfe von Step Functions einen Saga-Orchestrator erstellen können. Den vollständigen Code finden Sie im GitHubRepository für dieses Beispiel.

Aufgabendefinitionen

```
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);
```

Definitionen von Step Functions und Zustandsmaschine

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

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

GitHub Repository

Eine vollständige Implementierung der Beispielarchitektur für dieses Muster finden Sie im GitHub Repository unter https://github.com/aws-samples/ saga-orchestration-netcore-blog.

Blog-Referenzen

- Erstellen einer verteilten Serverless-Anwendung mithilfe des Saga-Orchestrierungs-Musters

Verwandter Inhalt

- Saga-Choreographie-Muster
- Transactional-Outbox-Muster

Videos

Im folgenden Video wird beschrieben, wie das Saga-Orchestrierungsmuster mithilfe AWS Step Functions von implementiert wird.

Scatter-Gather-Muster

Absicht

Das Scatter-Gather-Muster ist ein Nachrichtenweiterleitungsmuster, bei dem ähnliche oder verwandte Anfragen an mehrere Empfänger gesendet und deren Antworten mithilfe einer als Aggregator bezeichneten Komponente wieder zu einer einzigen Nachricht zusammengefasst werden. Dieses Muster trägt zur Parallelisierung bei, reduziert die Verarbeitungslatenz und verarbeitet asynchrone Kommunikation. Es ist einfach, das Scatter-Gather-Muster mithilfe eines synchronen Ansatzes zu implementieren, aber ein leistungsfähigerer Ansatz besteht darin, es als Nachrichtenrouting in asynchroner Kommunikation zu implementieren, entweder mit oder ohne Messaging-Dienst.

Motivation

Bei der Anwendungsverarbeitung kann eine Anfrage, deren sequentielle Verarbeitung viel Zeit in Anspruch nehmen kann, in mehrere Anfragen aufgeteilt werden, die parallel verarbeitet werden. Sie können Anfragen auch über API-Aufrufe an mehrere externe Systeme senden, um eine Antwort zu erhalten. Das Scatter-Gather-Muster ist nützlich, wenn Sie Eingaben aus mehreren Quellen benötigen. Scatter-Gather aggregiert die Ergebnisse, um Ihnen zu helfen, eine fundierte Entscheidung zu treffen oder die beste Antwort für die Anfrage auszuwählen.

Das Scatter-Gather-Muster besteht, wie der Name schon sagt, aus zwei Phasen:

- Die Scatter-Phase verarbeitet die Anforderungsnachricht und sendet sie parallel an mehrere Empfänger. Während dieser Phase verteilt die Anwendung Anfragen über das Netzwerk und läuft weiter, ohne auf sofortige Antworten zu warten.
- Während der Sammelphase sammelt die Anwendung die Antworten der Empfänger und filtert oder kombiniert sie zu einer einheitlichen Antwort. Wenn alle Antworten gesammelt wurden, können sie entweder zu einer einzigen Antwort zusammengefasst oder die beste Antwort für die weitere Verarbeitung ausgewählt werden.

Anwendbarkeit

Verwenden Sie das Scatter-Gather-Muster, wenn:

- Sie planen, Daten aus verschiedenen APIs zu aggregieren und zu konsolidieren, um eine genaue Antwort zu erhalten. Das Muster konsolidiert Informationen aus unterschiedlichen Quellen zu einem zusammenhängenden Ganzen. Ein Buchungssystem kann beispielsweise eine Anfrage an mehrere Empfänger richten, um Angebote von mehreren externen Partnern einzuholen.
- Dieselbe Anfrage muss gleichzeitig an mehrere Empfänger gesendet werden, um eine Transaktion abzuschließen. Sie können dieses Muster beispielsweise verwenden, um parallel Inventardaten abzufragen, um die Verfügbarkeit eines Produkts zu überprüfen.
- Sie möchten ein zuverlässiges und skalierbares System implementieren, in dem der Lastenausgleich durch die Verteilung von Anfragen auf mehrere Empfänger erreicht werden kann. Wenn ein Empfänger ausfällt oder einer hohen Auslastung ausgesetzt ist, können andere Empfänger weiterhin Anfragen bearbeiten.
- Sie möchten die Leistung optimieren, wenn Sie komplexe Abfragen implementieren, die mehrere Datenquellen einbeziehen. Sie können die Abfrage auf relevante Datenbanken verteilen, die Teilergebnisse zusammenfassen und sie zu einer umfassenden Antwort kombinieren.
- Sie implementieren eine Art der Map-Reduce-Verarbeitung, bei der die Datenanforderung zum Sharding und zur Replikation an mehrere Datenverarbeitungsendpunkte weitergeleitet wird. Teilergebnisse werden gefiltert und kombiniert, um die richtige Antwort zu erhalten.
- Bei schreibintensiven Arbeitslasten in Schlüsselwertdatenbanken möchten Sie Schreibvorgänge auf den Schlüsselbereich einer Partition verteilen. Der Aggregator liest die Ergebnisse, indem er die Daten in jedem Shard abfragt, und konsolidiert sie dann zu einer einzigen Antwort.

Fehler und Überlegungen

- Fehlertoleranz: Dieses Muster basiert auf mehreren Empfängern, die parallel arbeiten. Daher ist es wichtig, Fehler ordnungsgemäß zu behandeln. Um die Auswirkungen von Empfängerausfällen auf das Gesamtsystem zu minimieren, können Sie Strategien wie Redundanz, Replikation und Fehlererkennung implementieren.
- Scale-Out-Limits: Mit zunehmender Gesamtzahl der Verarbeitungsknoten steigt auch der damit verbundene Netzwerk-Overhead. Jede Anfrage, die eine Kommunikation über das Netzwerk beinhaltet, kann die Latenz erhöhen und sich negativ auf die Vorteile der Parallelisierung auswirken.
- Engpässe bei der Antwortzeit: Bei Vorgängen, bei denen alle Empfänger bearbeitet werden müssen, bevor die endgültige Verarbeitung abgeschlossen ist, wird die Leistung des Gesamtsystems durch die Antwortzeit des langsamsten Empfängers eingeschränkt.

- Teilweise Antworten: Wenn Anfragen an mehrere Empfänger verteilt werden, kann es bei einigen Empfängern zu einer Zeitüberschreitung kommen. In diesen Fällen sollte die Implementierung dem Kunden mitteilen, dass die Antwort unvollständig ist. Sie können die Details der Antwortaggregation auch mithilfe eines UI-Frontends anzeigen.
- Datenkonsistenz: Wenn Sie Daten für mehrere Empfänger verarbeiten, müssen Sie die Techniken zur Datensynchronisierung und Konfliktlösung sorgfältig abwägen, um sicherzustellen, dass die endgültigen aggregierten Ergebnisse korrekt und konsistent sind.

Implementierung

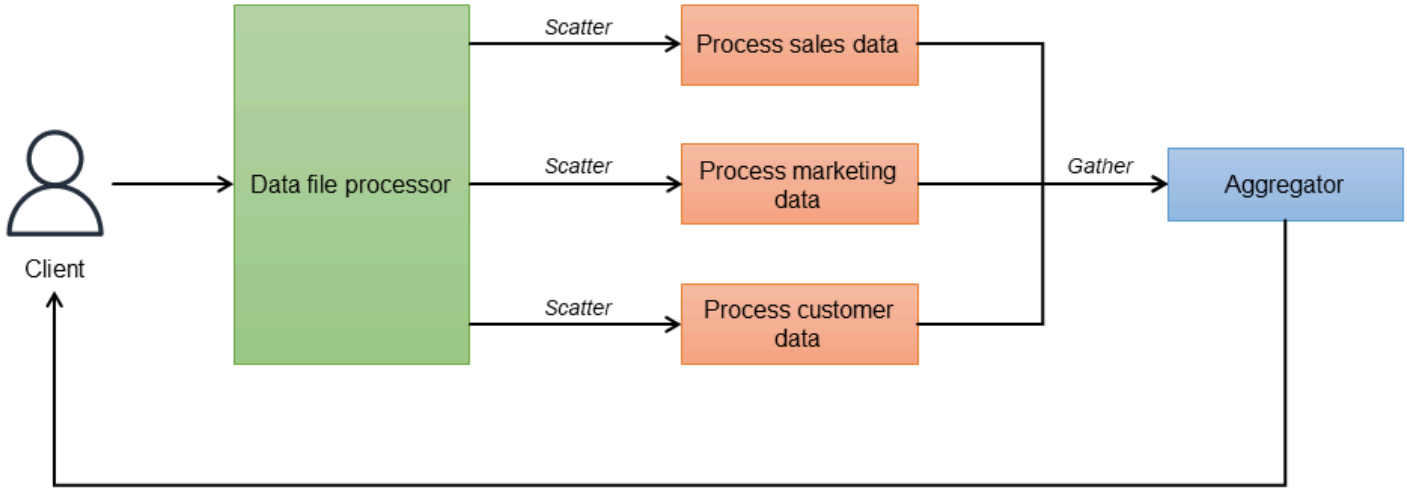
Hochrangige Architektur

Das Scatter-Gather-Muster verwendet einen Root-Controller, um Anfragen an Empfänger zu verteilen, die die Anfragen bearbeiten. Während der Scatter-Phase kann dieses Muster zwei Mechanismen nutzen, um Nachrichten an Empfänger zu senden:

- Streuung nach Verteilung: Die Anwendung verfügt über eine bekannte Liste von Empfängern, die aufgerufen werden müssen, um die Ergebnisse zu erhalten. Bei den Empfängern kann es sich um unterschiedliche Prozesse mit einzigartigen Funktionen oder um einen einzelnen Prozess handeln, der skaliert wurde, um die Verarbeitungslast zu verteilen. Wenn bei einem der Verarbeitungsknoten ein Timeout auftritt oder Verzögerungen bei der Reaktion auftreten, kann der Controller die Verarbeitung auf einen anderen Knoten umverteilen.
- Auktionsweise streuen: Die Anwendung sendet die Nachricht mithilfe eines Publish-Subscribe-Musters an interessierte Empfänger. In diesem Fall können die Empfänger die Nachricht abonnieren oder das Abonnement jederzeit kündigen.

Nach Verteilung streuen

Bei der Methode „Streuung nach Verteilung“ teilt der Root-Controller die eingehende Anfrage in unabhängige Aufgaben auf und weist sie verfügbaren Empfängern zu (die Scatter-Phase). Jeder Empfänger (Prozess, Container oder Lambda-Funktion) arbeitet unabhängig und parallel an seiner Berechnung und erzeugt einen Teil der Antwort. Wenn die Empfänger ihre Aufgaben abgeschlossen haben, senden sie ihre Antworten an einen Aggregator (die Sammelphase). Der Aggregator kombiniert die Teilantworten und gibt das Endergebnis an den Client zurück. Das folgende Diagramm veranschaulicht diesen Arbeitsablauf.



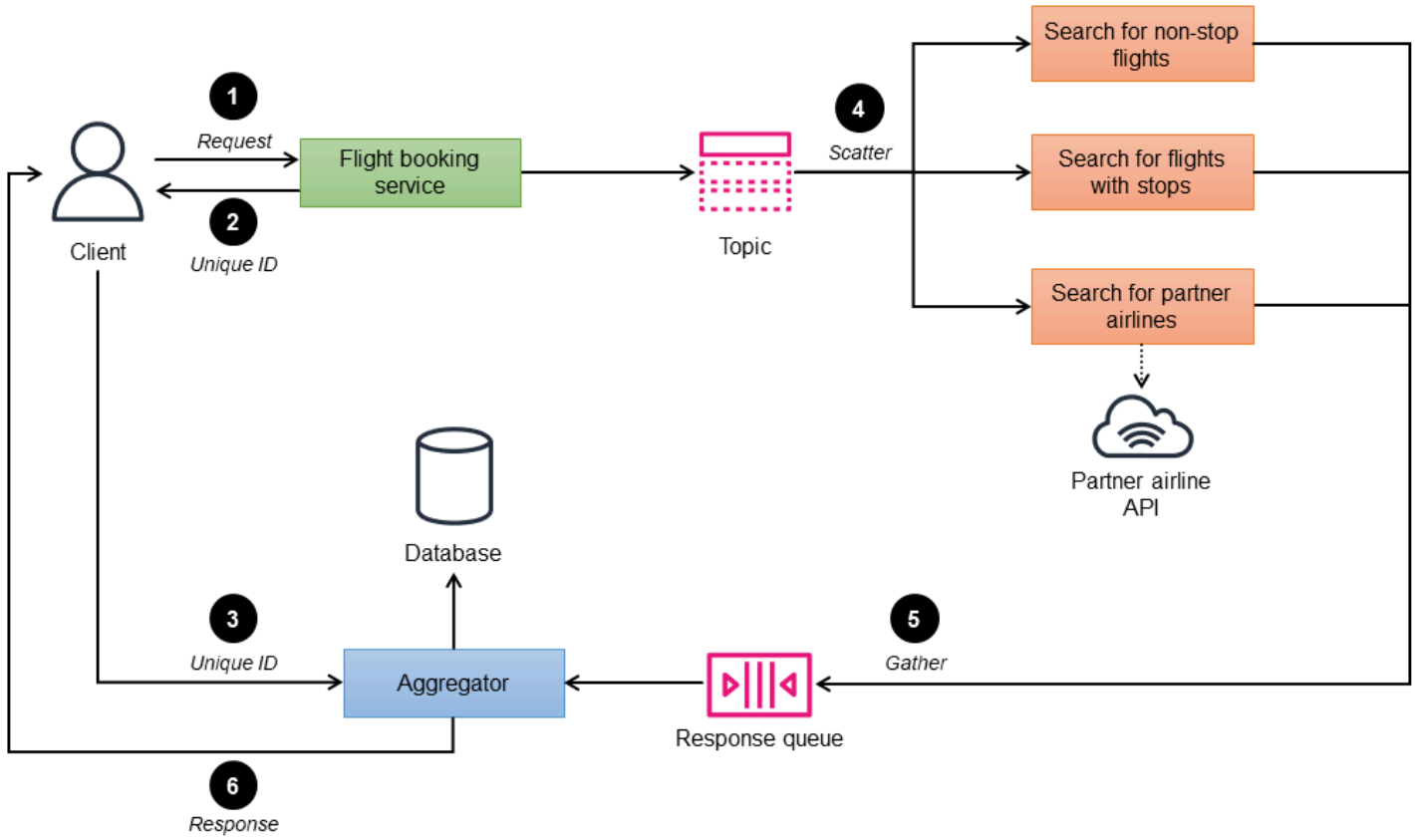
Der Controller (Datendateiprozessor) orchestriert den gesamten Satz von Aufrufen und kennt alle Buchungsendpunkte, die aufgerufen werden müssen. Es kann einen Timeout-Parameter so konfigurieren, dass Antworten ignoriert werden, die zu lange dauern. Wenn die Anfragen gesendet wurden, wartet der Aggregator auf die Antworten von jedem Endpunkt. Um Resilienz zu implementieren, kann jeder Microservice mit mehreren Instanzen für den Lastenausgleich bereitgestellt werden. Der Aggregator ruft die Ergebnisse ab, fasst sie zu einer einzigen Antwortnachricht zusammen und entfernt doppelte Daten vor der weiteren Verarbeitung. Die Antworten, bei denen das Timeout überschritten wird, werden ignoriert. Der Controller kann auch als Aggregator agieren, anstatt einen separaten Aggregatordienst zu verwenden.

Nach Auktion streuen

Wenn der Controller die Empfänger nicht kennt oder die Empfänger lose miteinander verknüpft sind, können Sie die Methode „Streuung nach Auktion“ verwenden. Bei dieser Methode abonnieren die Empfänger ein Thema und der Controller veröffentlicht die Anfrage zu dem Thema. Die Empfänger veröffentlichen die Ergebnisse in einer Antwortwarteschlange. Da der Stammcontroller die Empfänger nicht kennt, verwendet der Sammelvorgang einen Aggregator (ein anderes Nachrichtenmuster), um die Antworten zu sammeln und sie zu einer einzigen Antwortnachricht zusammenzufassen. Der Aggregator verwendet eine eindeutige ID, um eine Gruppe von Anfragen zu identifizieren.

In der folgenden Abbildung wird beispielsweise die Methode „Streuung nach Auktion“ verwendet, um einen Flugbuchungsservice für die Website einer Fluggesellschaft zu implementieren. Die Website ermöglicht es Benutzern, Flüge der eigenen Fluggesellschaft und der Fluggesellschaften ihrer Partner zu suchen und anzuzeigen. Außerdem muss der Status der Suche in Echtzeit angezeigt

werden. Der Flugbuchungsservice besteht aus drei Such-Microservices: Nonstop-Flüge, Flüge mit Zwischenstopps und Partnerfluggesellschaften. Die Suche nach Partnerfluggesellschaften ruft die API-Endpunkte des Partners auf, um die Antworten zu erhalten.

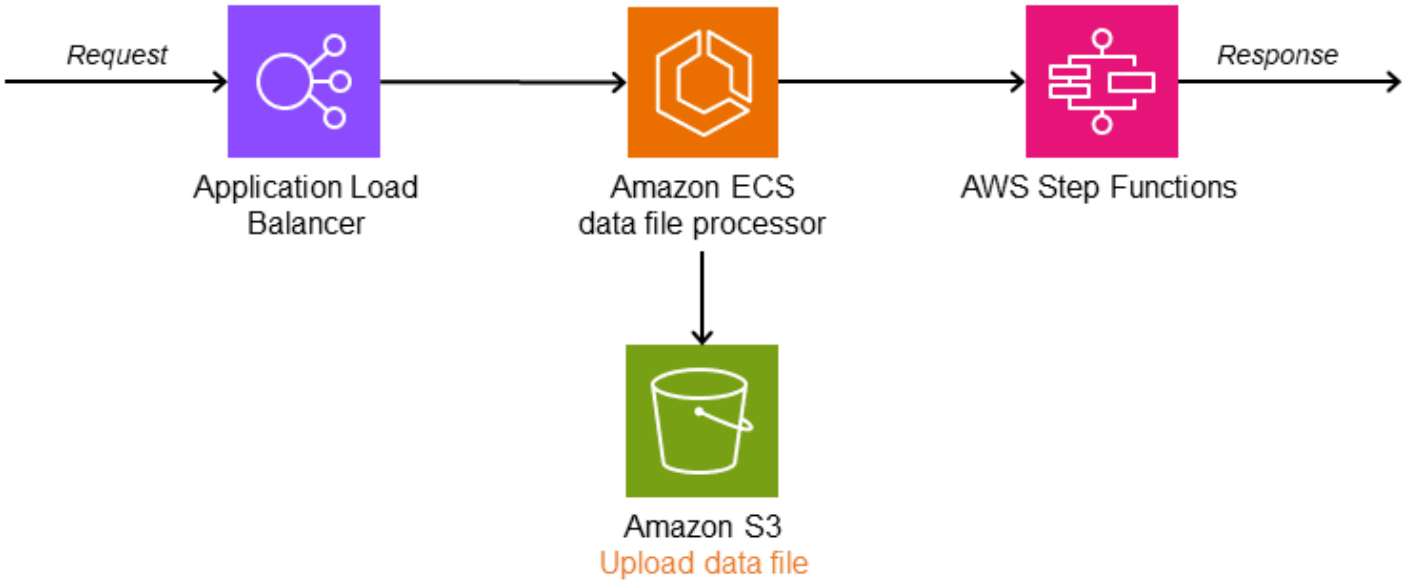


1. Der Flugbuchungsservice (Controller) verwendet die Suchkriterien als Eingabe des Kunden und bearbeitet und veröffentlicht die Anfrage zum Thema.
2. Der für die Verarbeitung Verantwortliche verwendet eine eindeutige ID, um jede Gruppe von Anfragen zu identifizieren.
3. Der Client sendet die eindeutige ID für Schritt 6 an den Aggregator.
4. Die Microservices für die Buchungssuche, die das Buchungsthema abonniert haben, erhalten die Anfrage.
5. Die Microservices bearbeiten die Anfrage und geben die Verfügbarkeit von Sitzplätzen für die angegebenen Suchkriterien an eine Antwortwarteschlange zurück.
6. Der Aggregator sammelt alle Antwortnachrichten, die in einer temporären Datenbank gespeichert sind, gruppiert die Flüge nach einer eindeutigen ID, erstellt eine einzige einheitliche Antwort und sendet sie an den Kunden zurück.

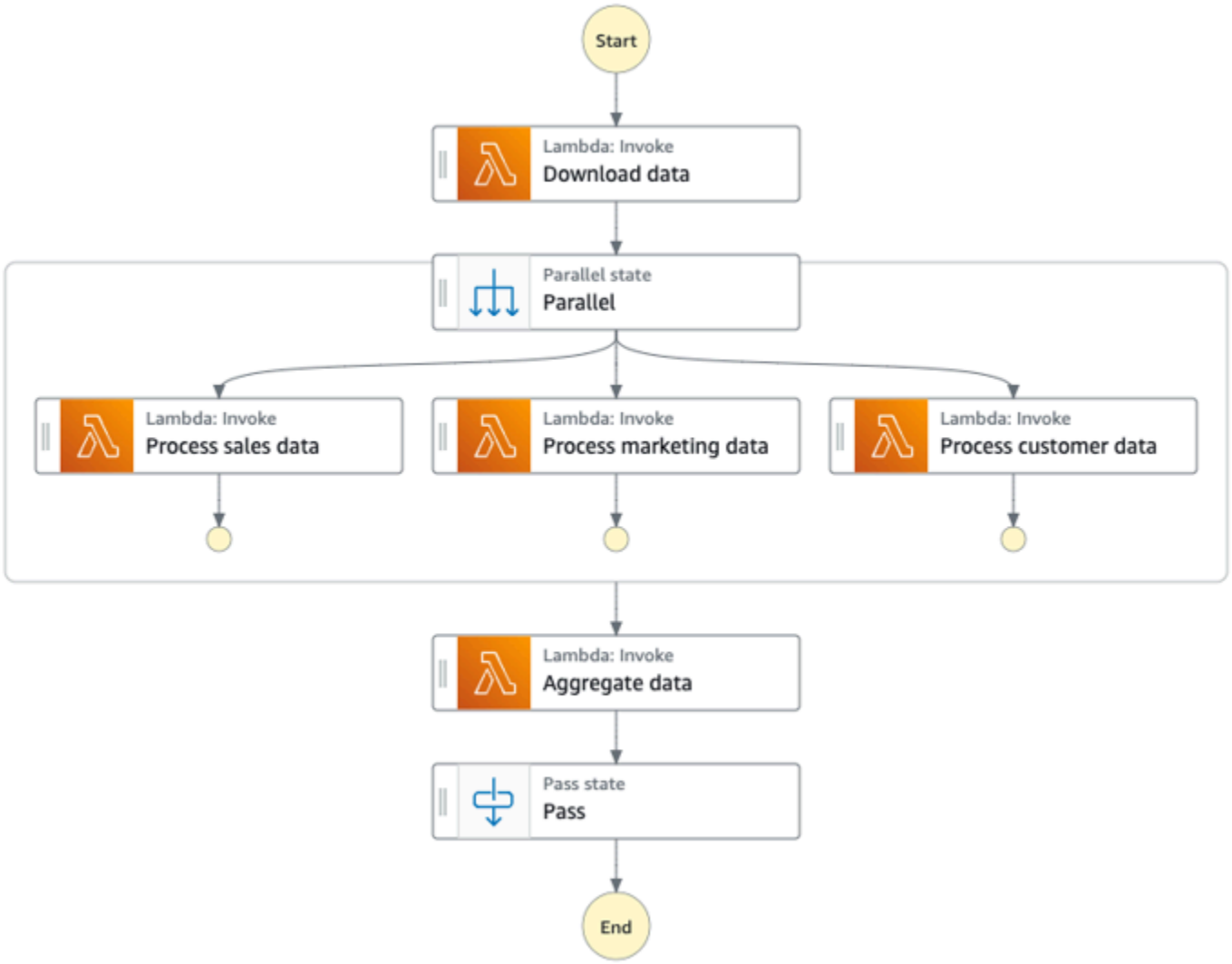
Implementierung unter Verwendung von AWS -Services

Streuung nach Verteilung

In der folgenden Architektur ist der Root-Controller ein Datendateiprozessor (Amazon ECS), der die eingehenden Anforderungsdaten in einzelne Amazon Simple Storage Service (Amazon S3) -Buckets aufteilt und einen AWS Step Functions Workflow startet. Der Workflow lädt die Daten herunter und initiiert die parallel Dateiverarbeitung. Der `Parallel` Status wartet darauf, dass alle Aufgaben eine Antwort zurückgeben. Eine AWS Lambda Funktion aggregiert die Daten und speichert sie wieder in Amazon S3.

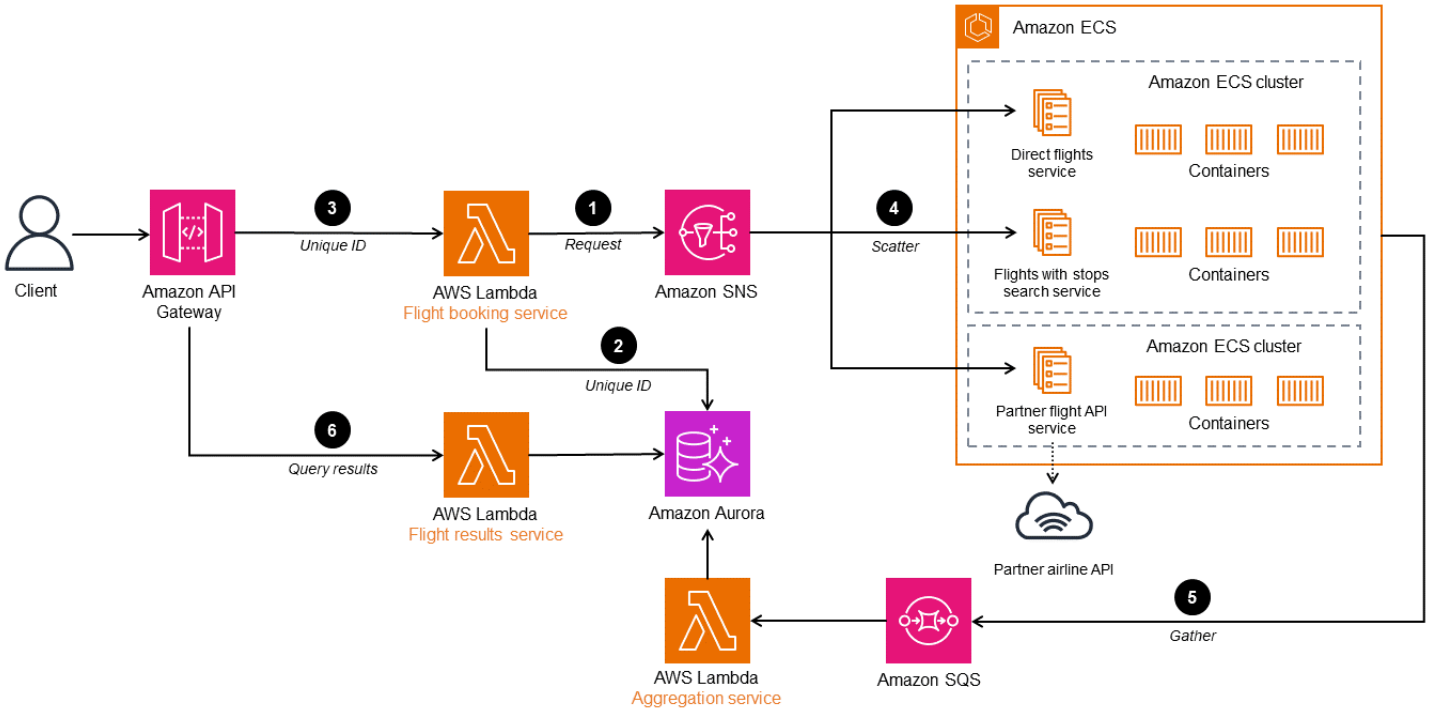


Das folgende Diagramm veranschaulicht den Step Functions Functions-Workflow mit dem `Parallel` Status.



Streuung nach Auktion

Das folgende Diagramm zeigt eine AWS Architektur für die Methode „Streuung nach Auktion“. Der Root-Controller-Flugbuchungsdienst verteilt die Flugsuchanfrage an mehrere Microservices. Ein Publish-Subscribe-Kanal wird mit Amazon Simple Notification Service (Amazon SNS) implementiert, einem verwalteten Messaging-Dienst für die Kommunikation. Amazon SNS unterstützt Nachrichten zwischen entkoppelten Microservice-Anwendungen oder direkte Kommunikation mit Benutzern. Sie können die Empfänger-Microservices auf Amazon Elastic Kubernetes Service (Amazon EKS) oder Amazon Elastic Container Service (Amazon ECS) bereitstellen, um die Verwaltung und Skalierbarkeit zu verbessern. Der Flugergebnis-Service sendet die Ergebnisse an den Kunden zurück. Es kann in AWS Lambda oder anderen Container-Orchestrierungsdiensten wie Amazon ECS oder Amazon EKS implementiert werden.



1. Der Flugbuchungsservice (Controller) verwendet die Suchkriterien als Eingabe vom Kunden und verarbeitet und veröffentlicht die Anfrage zum SNS-Thema.
2. Der Controller veröffentlicht die eindeutige ID in einer Amazon Aurora Aurora-Datenbank, um die Anfrage zu identifizieren.
3. Der Client sendet die eindeutige ID für Schritt 6 an den Client.
4. Die Microservices für die Buchungssuche, die das Buchungsthema abonniert haben, erhalten die Anfrage.
5. Die Microservices verarbeiten die Anfrage und geben die Verfügbarkeit von Sitzplätzen für die angegebenen Suchkriterien an eine Antwortwarteschlange in Amazon Simple Queue Service (Amazon SQS) zurück. Der Aggregator sammelt alle Antwortnachrichten und speichert sie in einer temporären Datenbank.
6. Der Flugergebnisdienst gruppiert die Flüge nach einer eindeutigen ID, erstellt eine einzige einheitliche Antwort und sendet sie an den Kunden zurück.

Wenn Sie dieser Architektur eine weitere Airline-Suche hinzufügen möchten, fügen Sie einen Microservice hinzu, der das SNS-Thema abonniert und in der SQS-Warteschlange veröffentlicht.

Zusammenfassend lässt sich sagen, dass das Scatter-Gather-Muster es verteilten Systemen ermöglicht, eine effiziente Parallelisierung zu erreichen, die Latenz zu reduzieren und asynchrone Kommunikation nahtlos zu verarbeiten.

GitHub Repositorium

Eine vollständige Implementierung der Beispielarchitektur für dieses Muster finden Sie im GitHub Repository unter https://github.com/aws-samples/ asynchronous-messaging-workshop /tree/master/code/lab-3.

Workshop

- Versuchslabor im Workshop „Decoupled Microservices“

Blog-Referenzen

- Muster der Anwendungsintegration für Microservices

Verwandter Inhalt

- Muster „Veröffentlichen — Abonnieren“

Würger-Feigenmuster

Absicht

Das Strangler-Fig-Muster hilft dabei, eine monolithische Anwendung schrittweise auf eine Microservices-Architektur zu migrieren, wodurch das Transformationsrisiko und die Betriebsunterbrechung reduziert werden.

Motivation

Monolithische Anwendungen werden so entwickelt, dass sie den Großteil ihrer Funktionalität in einem einzigen Prozess oder Container bereitstellen. Der Code ist eng miteinander verknüpft. Daher müssen Anwendungsänderungen gründlich erneut getestet werden, um Regressionsprobleme zu vermeiden. Die Änderungen können nicht isoliert getestet werden, was sich auf die Zykluszeit auswirkt. Da die Anwendung um mehr Funktionen erweitert wird, kann eine hohe Komplexität dazu führen, dass mehr Zeit für die Wartung aufgewendet wird, die Markteinführungszeit verlängert wird und folglich die Produktinnovation verlangsamt wird.

Wenn die Anwendung an Größe zunimmt, erhöht dies die kognitive Belastung des Teams und kann zu unklaren Eigentümergrenzen führen. Eine Skalierung einzelner Funktionen auf der Grundlage der Auslastung ist nicht möglich — die gesamte Anwendung muss skaliert werden, um Spitzenlasten zu bewältigen. Mit zunehmendem Alter der Systeme kann die Technologie veraltet sein, was die Supportkosten in die Höhe treibt. Monolithische Legacy-Anwendungen folgen bewährten Methoden, die zum Zeitpunkt der Entwicklung verfügbar waren und nicht für den Vertrieb konzipiert waren.

Wenn eine monolithische Anwendung in eine Microservices-Architektur migriert wird, kann sie in kleinere Komponenten aufgeteilt werden. Diese Komponenten können unabhängig voneinander skaliert werden, können unabhängig voneinander veröffentlicht werden und können einzelnen Teams gehören. Dies führt zu einer höheren Änderungsgeschwindigkeit, da Änderungen lokalisiert sind und schnell getestet und veröffentlicht werden können. Änderungen haben einen geringeren Wirkungsbereich, da die Komponenten lose miteinander verknüpft sind und einzeln eingesetzt werden können.

Einen Monolithen vollständig durch eine Microservices-Anwendung zu ersetzen, indem der Code neu geschrieben oder umgestaltet wird, ist ein großes Unterfangen und ein großes Risiko. Eine Big-Bang-Migration, bei der der Monolith in einem einzigen Vorgang migriert wird, birgt Transformationsrisiken

und Geschäftsunterbrechungen. Während die Anwendung überarbeitet wird, ist es extrem schwierig oder sogar unmöglich, neue Funktionen hinzuzufügen.

Eine Möglichkeit, dieses Problem zu lösen, ist die Verwendung des Würgerfeigenmusters, das von Martin Fowler eingeführt wurde. Dieses Muster beinhaltet die Umstellung auf Microservices, indem Funktionen schrittweise extrahiert und eine neue Anwendung rund um das bestehende System erstellt werden. Die Funktionen im Monolithen werden schrittweise durch Microservices ersetzt, und Anwendungsbenutzer können die neu migrierten Funktionen schrittweise verwenden. Wenn alle Funktionen auf das neue System übertragen wurden, kann die monolithische Anwendung problemlos außer Betrieb genommen werden.

Anwendbarkeit

Verwenden Sie das Würgerfeigenmuster, wenn:

- Sie möchten Ihre monolithische Anwendung schrittweise auf eine Microservices-Architektur migrieren.
- Ein Big-Bang-Migrationsansatz ist aufgrund der Größe und Komplexität des Monolithen riskant.
- Das Unternehmen möchte neue Funktionen hinzufügen und kann es kaum erwarten, bis die Transformation abgeschlossen ist.
- Endbenutzer müssen während der Transformation nur minimal beeinträchtigt werden.

Probleme und Überlegungen

- Zugriff auf die Codebasis: Um das Strangler Fig-Muster zu implementieren, müssen Sie Zugriff auf die Codebasis der Monolith-Anwendung haben. Da Funktionen aus dem Monolithen migriert werden, müssen Sie kleinere Codeänderungen vornehmen und innerhalb des Monolithen eine Antikorruptionsebene implementieren, um Anrufe an neue Microservices weiterzuleiten. Ohne Zugriff auf die Codebasis können Sie keine Anrufe abfangen. Der Zugriff auf die Codebasis ist auch für die Umleitung eingehender Anfragen von entscheidender Bedeutung. Möglicherweise ist ein gewisses Code-Refactoring erforderlich, damit die Proxyschicht die Aufrufe für migrierte Funktionen abfangen und an Microservices weiterleiten kann.
- Unklarer Bereich: Die vorzeitige Zerlegung von Systemen kann kostspielig sein, insbesondere wenn die Domäne nicht klar ist und es möglich ist, dass die Dienstgrenzen falsch festgelegt werden. Domain-Driven Design (DDD) ist ein Mechanismus zum Verständnis der Domäne, und Event Storming ist eine Technik zur Bestimmung von Domänengrenzen.

- Identifizierung von Microservices: Sie können DDD als wichtiges Tool zur Identifizierung von Microservices verwenden. Um Microservices zu identifizieren, achten Sie auf die natürlichen Unterteilungen zwischen den Serviceklassen. Viele Dienste werden ihr eigenes Datenzugriffsobjekt besitzen und sich leicht entkoppeln lassen. Dienste mit verwandter Geschäftslogik und Klassen, die keine oder nur wenige Abhängigkeiten haben, eignen sich gut für Microservices. Sie können Code umgestalten, bevor Sie den Monolithen aufbrechen, um eine enge Kopplung zu verhindern. Sie sollten auch die Compliance-Anforderungen, den Release-Rhythmus, den geografischen Standort der Teams, Skalierungsanforderungen, anwendungsfallorientierte Technologieanforderungen und die kognitive Belastung der Teams berücksichtigen.
- Antikorruptionsebene: Während des Migrationsprozesses, wenn die Funktionen innerhalb des Monolithen die Funktionen aufrufen müssen, die als Microservices migriert wurden, sollten Sie eine Antikorruptionsebene (ACL) implementieren, die jeden Aufruf an den entsprechenden Microservice weiterleitet. Um bestehende Anrufer innerhalb des Monolithen zu entkoppeln und Änderungen an ihnen zu verhindern, fungiert die ACL als Adapter oder Fassade, die die Aufrufe in die neuere Schnittstelle konvertiert. Dies wird im Abschnitt Implementierung des ACL-Patterns weiter oben in diesem Handbuch ausführlich behandelt.
- Ausfall der Proxyschicht: Während der Migration fängt eine Proxyschicht die Anfragen ab, die an die monolithische Anwendung gehen, und leitet sie entweder an das Altsystem oder an das neue System weiter. Diese Proxyschicht kann jedoch zu einer einzigen Fehlerquelle oder zu einem Leistungsengpass werden.
- Komplexität der Anwendung: Große Monolithen profitieren am meisten vom Strangler-Feigen-Muster. Bei kleinen Anwendungen, bei denen die Komplexität eines vollständigen Refactorings gering ist, kann es effizienter sein, die Anwendung in einer Microservices-Architektur neu zu schreiben, anstatt sie zu migrieren.
- Serviceinteraktionen: Microservices können synchron oder asynchron kommunizieren. Wenn synchrone Kommunikation erforderlich ist, sollten Sie abwägen, ob die Timeouts zu einer Auslastung der Verbindung oder des Threadpools führen können, was zu Leistungsproblemen der Anwendung führen kann. Verwenden Sie in solchen Fällen das Circuit-Breaker-Muster, um bei Vorgängen, bei denen es wahrscheinlich ist, dass sie über einen längeren Zeitraum ausfallen, einen sofortigen Ausfall zu melden. Asynchrone Kommunikation kann mithilfe von Ereignissen und Nachrichtenwarteschlangen erreicht werden.
- Datenaggregation: In einer Microservices-Architektur werden Daten über Datenbanken verteilt. Wenn Datenaggregation erforderlich ist, können Sie dies AWS AppSyncim Frontend oder das CQRS-Muster (Command Query Responsibility Segregation) im Backend verwenden.

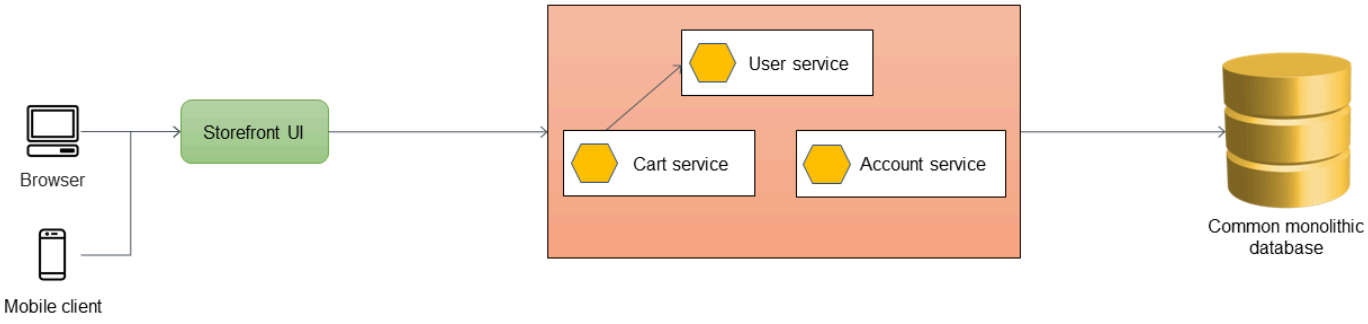
- Datenkonsistenz: Die Microservices besitzen ihren eigenen Datenspeicher, und die monolithische Anwendung kann diese Daten möglicherweise auch verwenden. Um die gemeinsame Nutzung zu ermöglichen, können Sie den Datenspeicher der neuen Microservices mithilfe einer Warteschlange und eines Agenten mit der Datenbank der monolithischen Anwendung synchronisieren. Dies kann jedoch zu Datenredundanz und letztendlich zu Konsistenz zwischen zwei Datenspeichern führen. Wir empfehlen daher, dies als taktische Lösung zu behandeln, bis Sie eine langfristige Lösung wie einen Data Lake einrichten können.

Implementierung

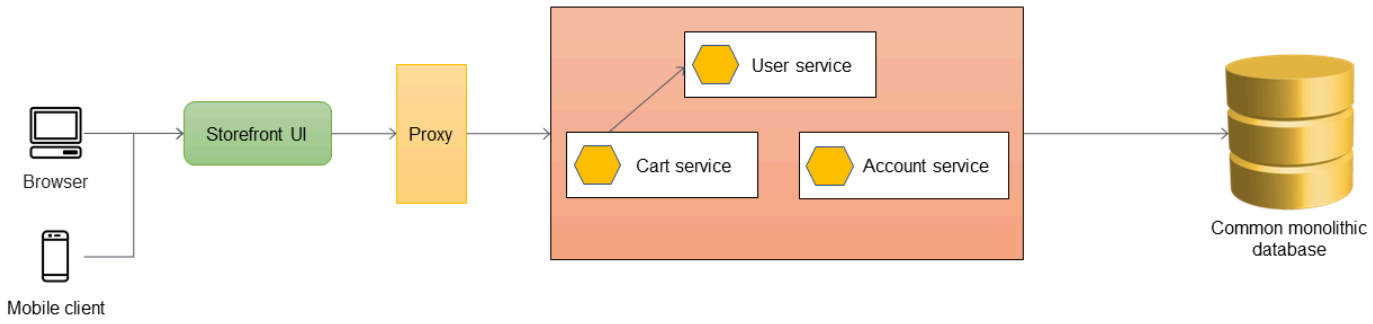
Nach dem Muster der Würgerfeige ersetzen Sie eine bestimmte Funktionalität durch einen neuen Dienst oder eine neue Anwendung, eine Komponente nach der anderen. Eine Proxyschicht fängt Anfragen ab, die an die monolithische Anwendung gehen, und leitet sie entweder an das Altsystem oder an das neue System weiter. Da die Proxyschicht Benutzer zur richtigen Anwendung weiterleitet, können Sie dem neuen System Funktionen hinzufügen und gleichzeitig sicherstellen, dass der Monolith weiterhin funktioniert. Das neue System ersetzt schließlich alle Funktionen des alten Systems, und Sie können es außer Betrieb nehmen.

Architektur auf hohem Niveau

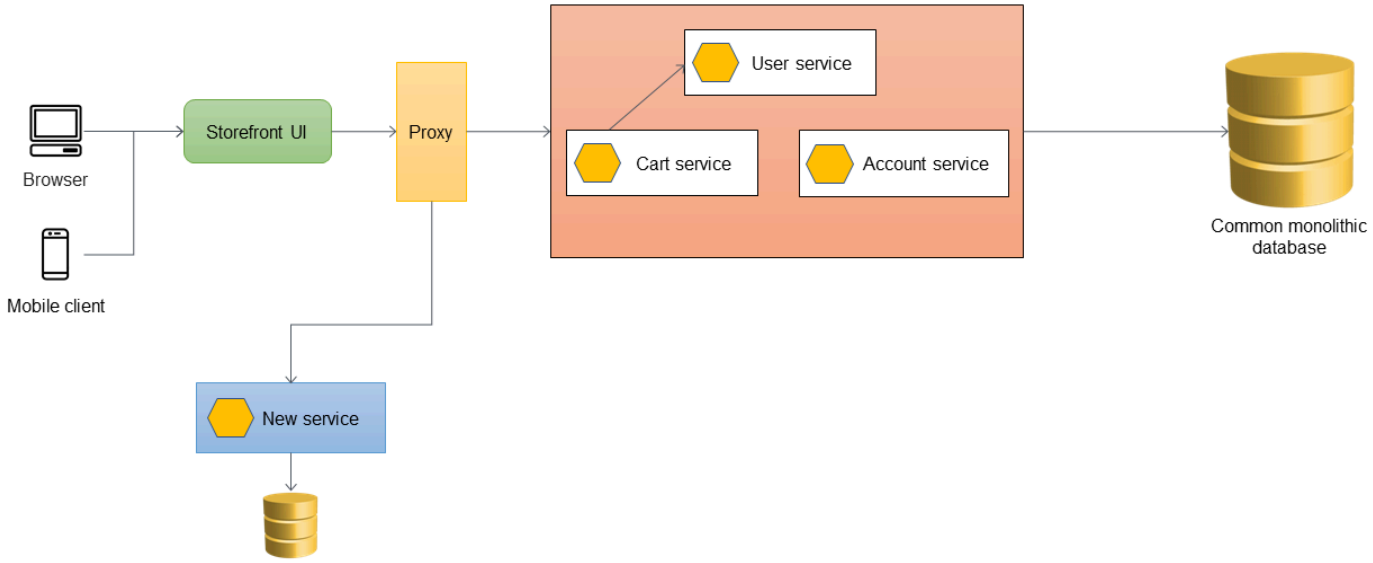
In der folgenden Abbildung verfügt eine monolithische Anwendung über drei Dienste: Benutzerservice, Warenkorbservice und Kontodienst. Der Warenkorb-Service hängt vom Benutzerdienst ab, und die Anwendung verwendet eine monolithische relationale Datenbank.



Der erste Schritt besteht darin, eine Proxyschicht zwischen der Storefront-Benutzeroberfläche und der monolithischen Anwendung hinzuzufügen. Zu Beginn leitet der Proxy den gesamten Datenverkehr an die monolithische Anwendung weiter.

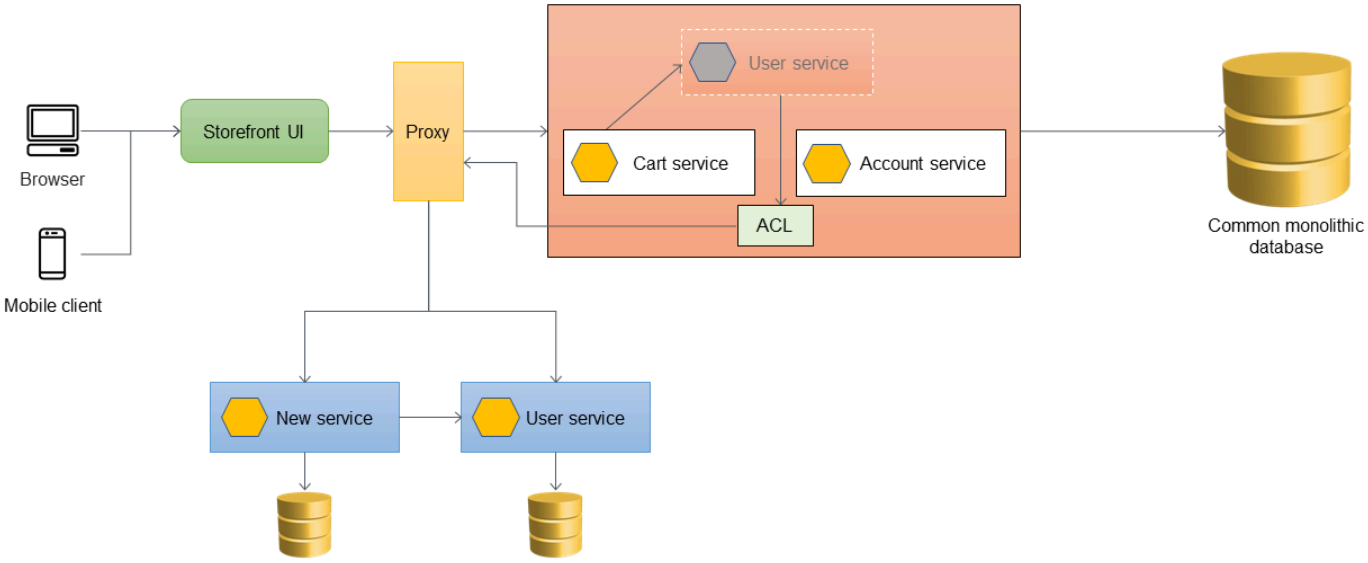


Wenn Sie Ihrer Anwendung neue Funktionen hinzufügen möchten, implementieren Sie diese als neue Microservices, anstatt dem vorhandenen Monolithen Funktionen hinzuzufügen. Sie beheben jedoch weiterhin Fehler im Monolithen, um die Stabilität der Anwendung zu gewährleisten. In der folgenden Abbildung leitet die Proxyschicht die Aufrufe auf der Grundlage der API-URL an den Monolith oder an den neuen Microservice weiter.



Hinzufügen einer Antikorruptionsebene

In der folgenden Architektur wurde der Benutzerdienst auf einen Microservice migriert. Der Cart-Service ruft den Benutzerservice auf, aber die Implementierung ist innerhalb des Monolithen nicht mehr verfügbar. Außerdem stimmt die Schnittstelle des neu migrierten Dienstes möglicherweise nicht mit der vorherigen Schnittstelle innerhalb der monolithischen Anwendung überein. Um diese Änderungen zu beheben, implementieren Sie eine ACL. Wenn während des Migrationsprozesses die Funktionen innerhalb des Monolithen die Funktionen aufrufen müssen, die als Microservices migriert wurden, konvertiert die ACL die Aufrufe an die neue Schnittstelle und leitet sie an den entsprechenden Microservice weiter.



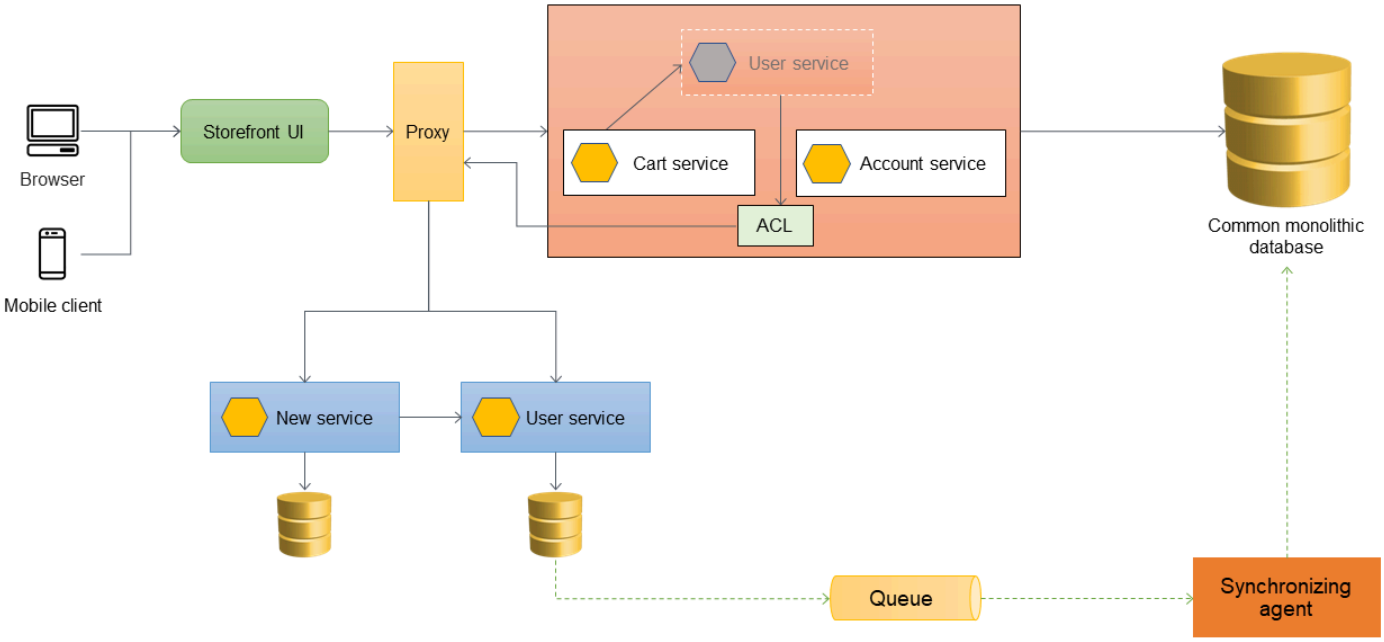
Sie können die ACL innerhalb der monolithischen Anwendung als Klasse implementieren, die für den Service spezifisch ist, der migriert wurde, `UserServiceFacade` z. B. oder. `UserServiceAdapter` Die ACL muss außer Betrieb genommen werden, nachdem alle abhängigen Dienste in die Microservices-Architektur migriert wurden.

Wenn Sie die ACL verwenden, ruft der Cart-Service den Benutzerservice weiterhin innerhalb des Monolithen auf, und der Benutzerdienst leitet den Aufruf über die ACL an den Microservice weiter. Der Warenkorb-Service sollte den Benutzerservice trotzdem aufrufen, ohne sich der Microservice-Migration bewusst zu sein. Diese lose Kopplung ist erforderlich, um Regression und Geschäftsunterbrechungen zu reduzieren.

Handhabung der Datensynchronisierung

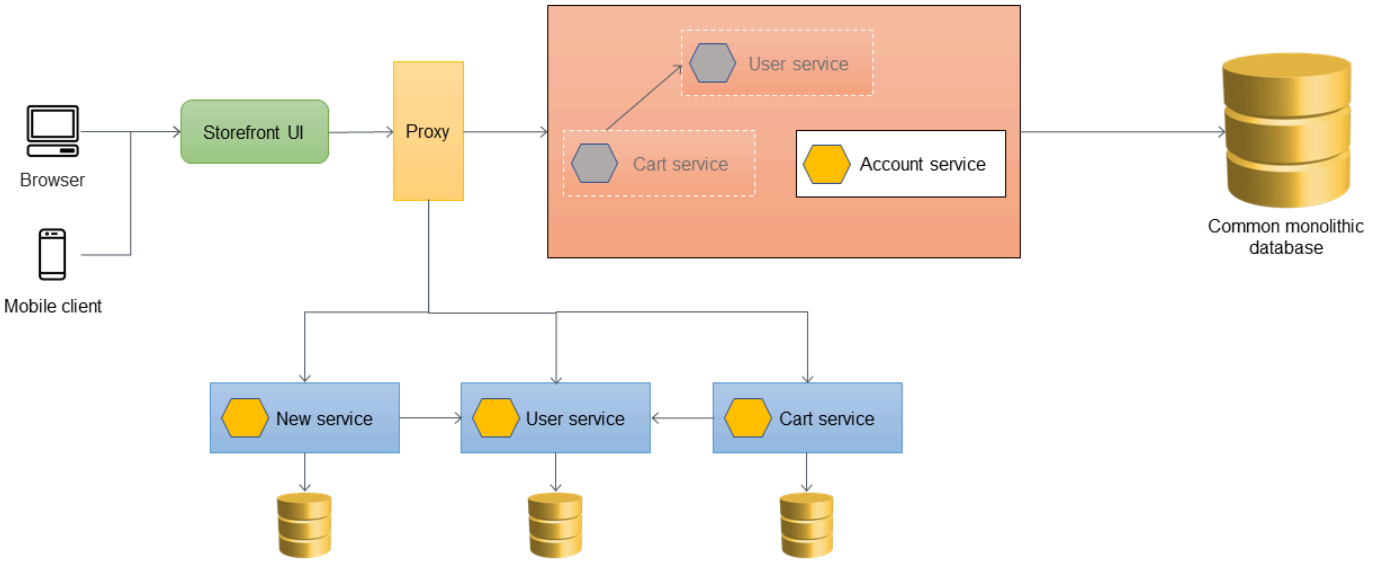
Es hat sich bewährt, dass der Microservice Eigentümer seiner Daten sein sollte. Der Benutzerdienst speichert seine Daten in seinem eigenen Datenspeicher. Möglicherweise muss er Daten mit der monolithischen Datenbank synchronisieren, um Abhängigkeiten wie die Berichterstattung zu handhaben und nachgelagerte Anwendungen zu unterstützen, die noch nicht bereit sind, direkt auf die Microservices zuzugreifen. Die monolithische Anwendung benötigt möglicherweise auch die Daten für andere Funktionen und Komponenten, die noch nicht auf Microservices migriert wurden. Daher ist eine Datensynchronisation zwischen dem neuen Microservice und dem Monolith erforderlich. Um die Daten zu synchronisieren, können Sie einen Synchronisierungsagenten zwischen dem Benutzer-Microservice und der monolithischen Datenbank einrichten, wie in der folgenden Abbildung dargestellt. Der Benutzer-Microservice sendet bei jeder Aktualisierung seiner Datenbank ein Ereignis an die Warteschlange. Der Synchronisierungsagent überwacht

die Warteschlange und aktualisiert die monolithische Datenbank kontinuierlich. Die Daten in der monolithischen Datenbank sind letztendlich konsistent für die Daten, die synchronisiert werden.

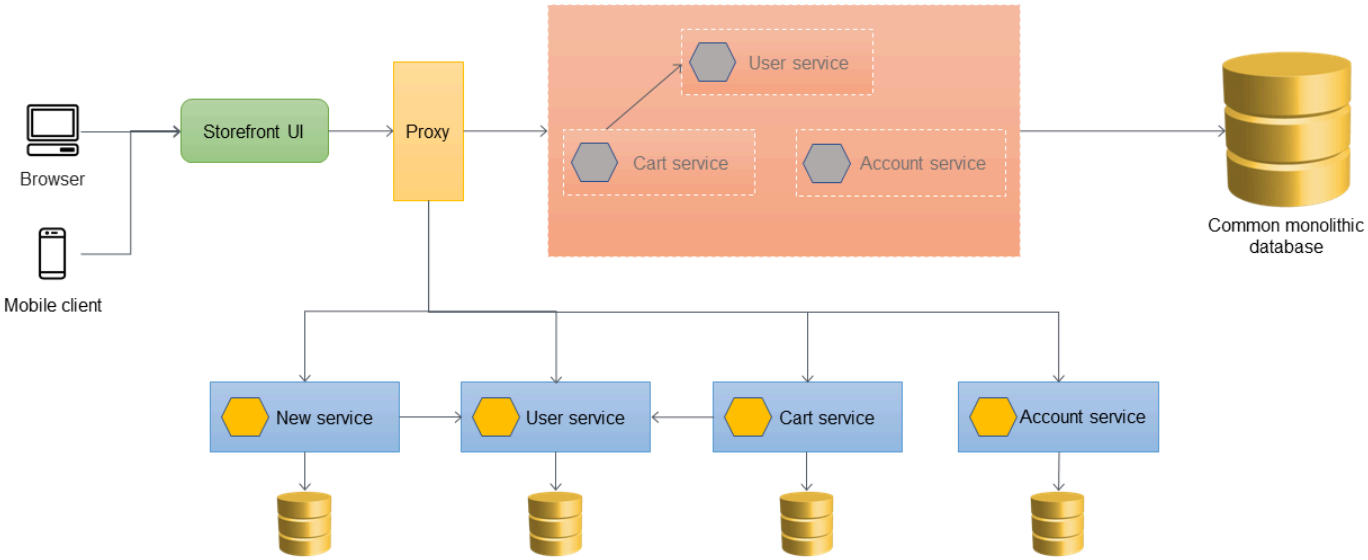


Zusätzliche Dienste werden migriert

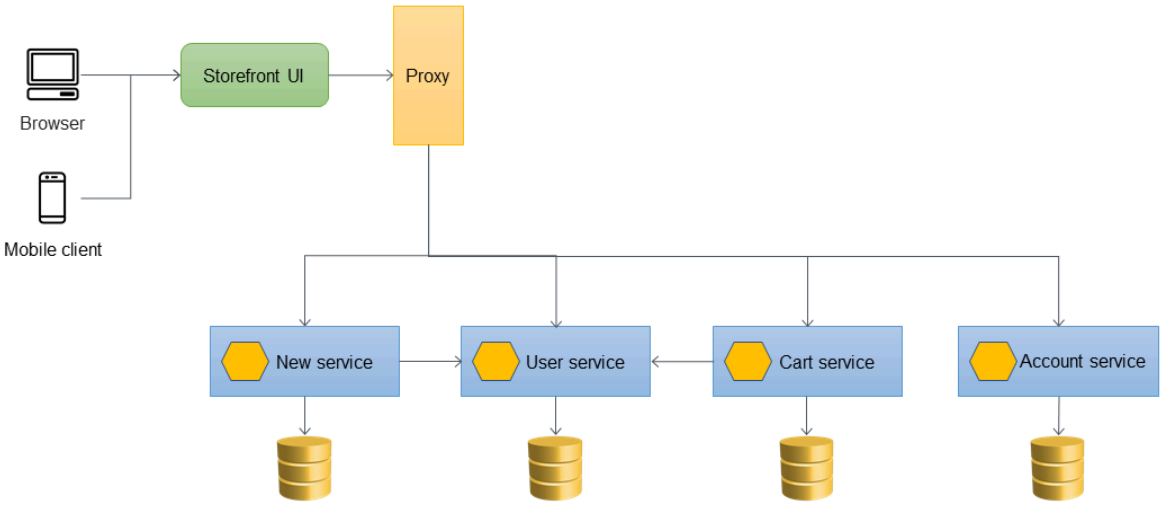
Wenn der Cart-Dienst aus der monolithischen Anwendung migriert wird, wird sein Code dahingehend überarbeitet, dass der neue Service direkt aufgerufen wird, sodass die ACL diese Aufrufe nicht mehr weiterleitet. Das folgende Diagramm veranschaulicht diese Architektur.



Das folgende Diagramm zeigt den endgültigen Zustand, in dem alle Dienste aus dem Monolithen migriert wurden und nur noch das Skelett des Monolithen übrig ist. Historische Daten können in Datenspeicher migriert werden, die einzelnen Diensten gehören. Die ACL kann entfernt werden, und der Monolith ist zu diesem Zeitpunkt bereit, außer Betrieb genommen zu werden.



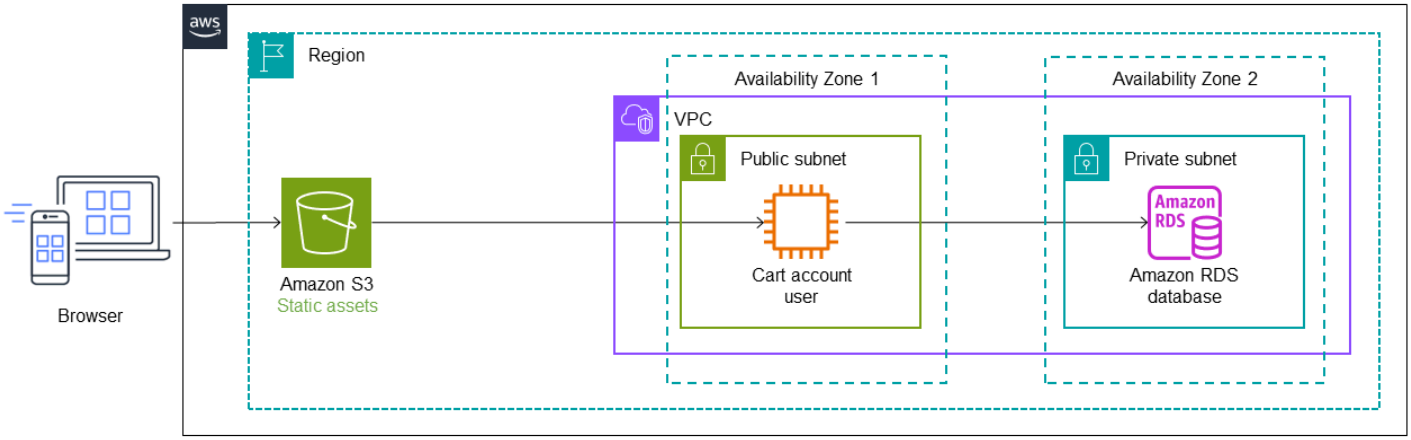
Das folgende Diagramm zeigt die endgültige Architektur nach der Außerbetriebnahme der monolithischen Anwendung. Sie können die einzelnen Microservices je nach den Anforderungen Ihrer Anwendung über eine ressourcenbasierte URL (z. B.`http://www.storefront.com/user`) oder über eine eigene Domain (z. B.`http://user.storefront.com`) hosten. Weitere Informationen zu den wichtigsten Methoden zur Bereitstellung von HTTP-APIs für Upstream-Verbraucher mithilfe von Hostnamen und Pfaden finden Sie im Abschnitt API-Routingmuster.



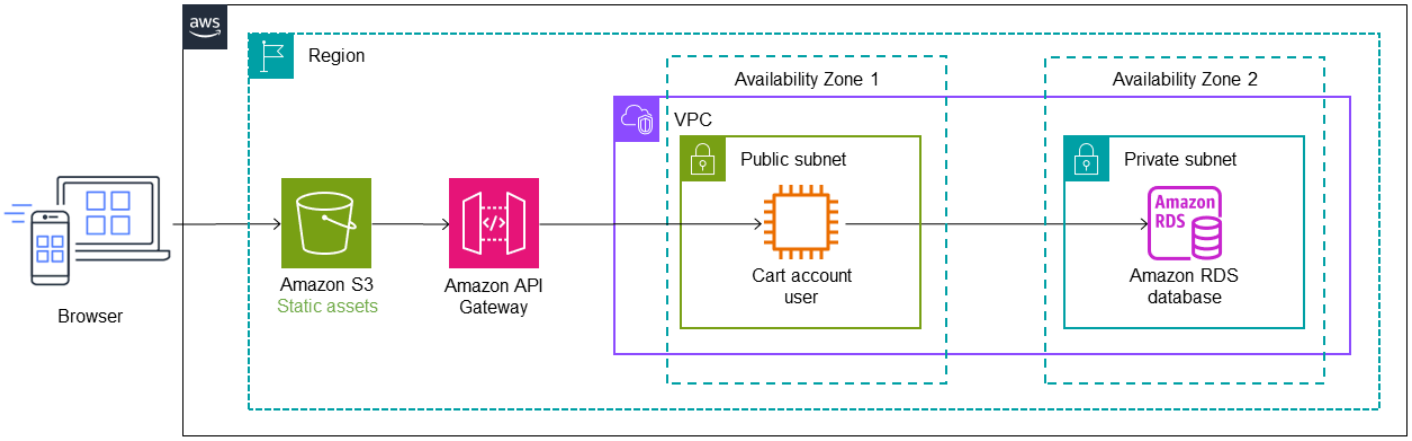
Implementierung mithilfe von Diensten AWS

API Gateway als Anwendungsproxy verwenden

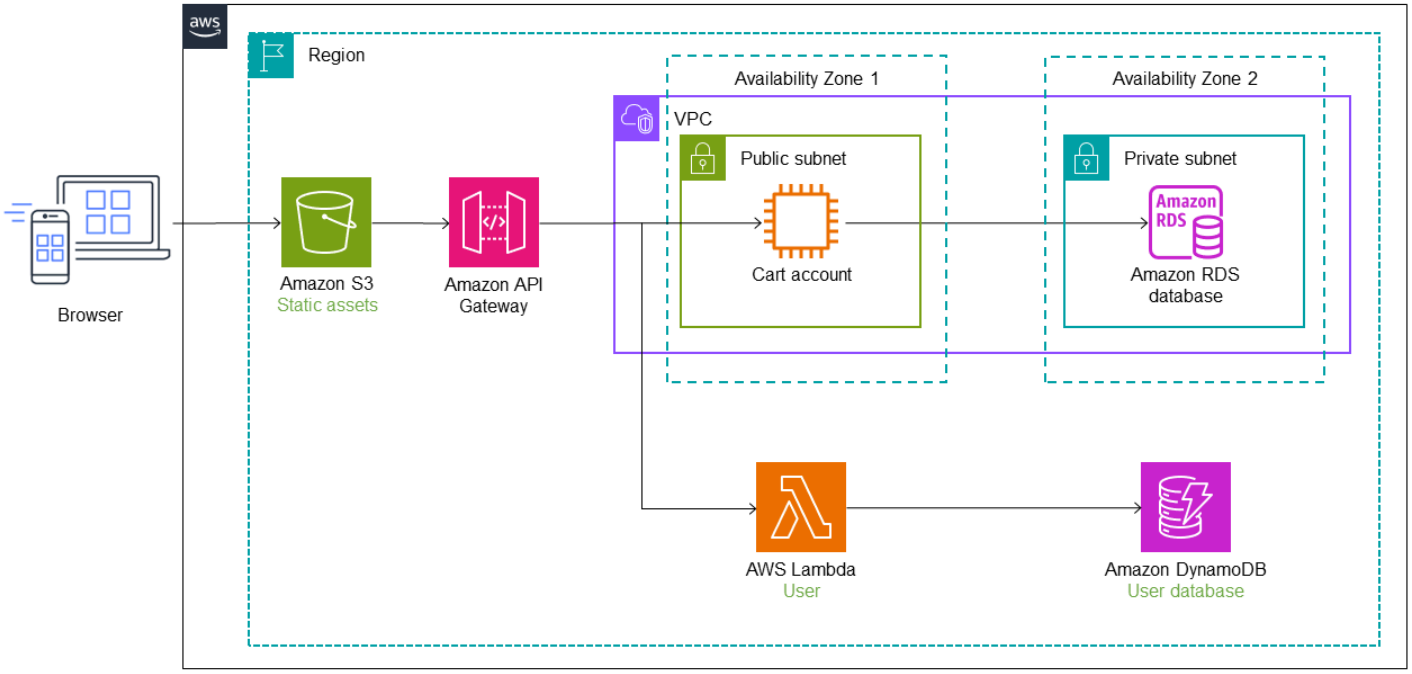
Das folgende Diagramm zeigt den Ausgangszustand der monolithischen Anwendung. Nehmen wir an, es wurde AWS mithilfe einer lift-and-shift Strategie migriert, sodass es auf einer Amazon Elastic Compute Cloud (Amazon EC2) -Instance läuft und eine Amazon Relational Database Service (Amazon RDS) -Datenbank verwendet. Der Einfachheit halber verwendet die Architektur eine einzige Virtual Private Cloud (VPC) mit einem privaten und einem öffentlichen Subnetz. Nehmen wir an, dass die Microservices zunächst innerhalb desselben bereitgestellt werden. AWS-Konto (Die bewährte Methode in Produktionsumgebungen ist die Verwendung einer Architektur mit mehreren Konten, um die Unabhängigkeit der Bereitstellung zu gewährleisten.) Die EC2-Instance befindet sich in einer einzigen Availability Zone im öffentlichen Subnetz, und die RDS-Instance befindet sich in einer einzigen Availability Zone im privaten Subnetz. Amazon Simple Storage Service (Amazon S3) speichert statische Ressourcen wie die JavaScript, CSS- und React-Dateien für die Website.



In der folgenden Architektur AWS Migration Hub Refactor Spaceswird Amazon API Gateway vor der monolithischen Anwendung bereitgestellt. Refactor Spaces erstellt eine Refactoring-Infrastruktur in Ihrem Konto, und API Gateway fungiert als Proxyschicht für die Weiterleitung von Aufrufen an den Monolithen. Anfänglich werden alle Aufrufe über die Proxyschicht an die monolithische Anwendung weitergeleitet. Wie bereits erwähnt, können Proxyschichten zu einer einzigen Fehlerquelle werden. Die Verwendung von API Gateway als Proxy mindert das Risiko jedoch, da es sich um einen serverlosen Multi-AZ-Dienst handelt.

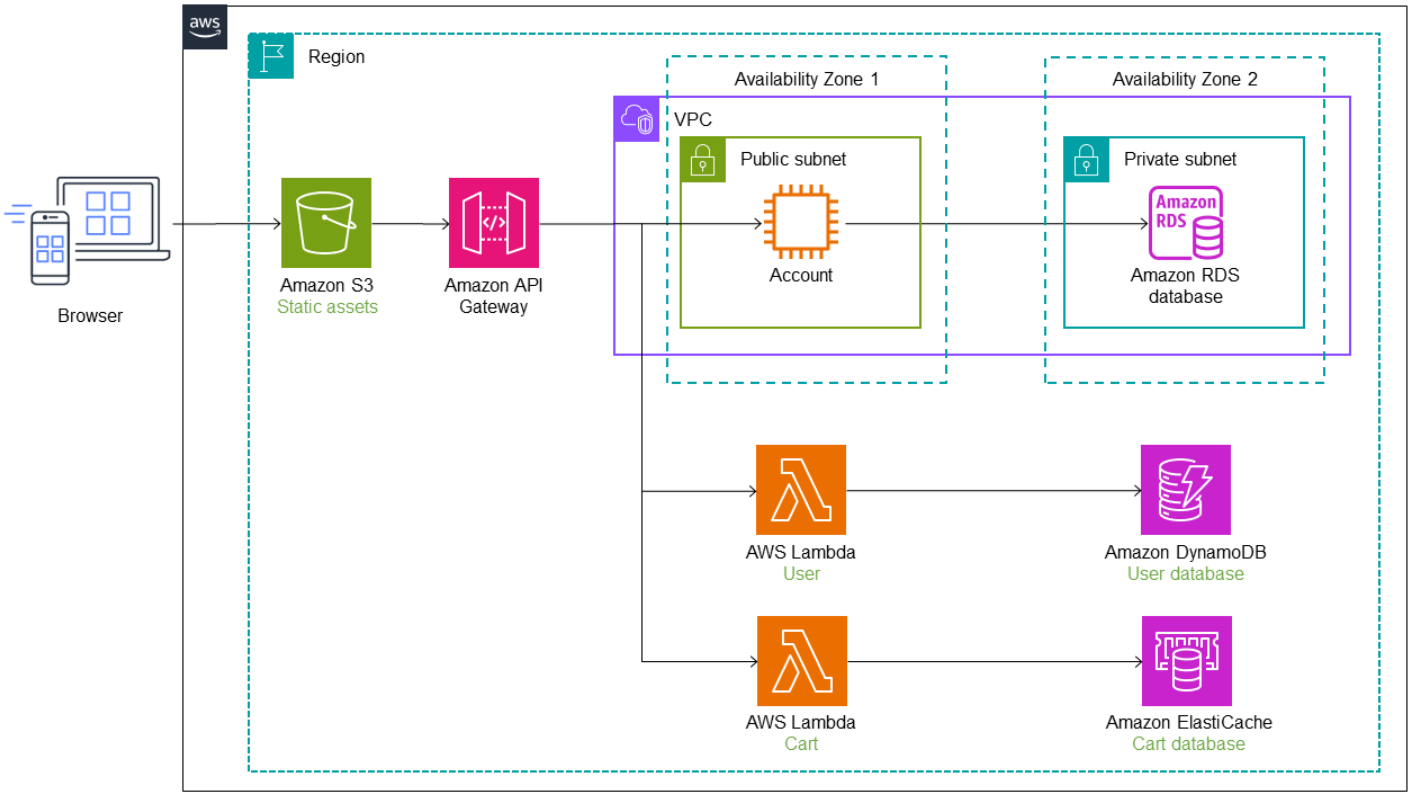


Der Benutzerservice wird in eine Lambda-Funktion migriert, und seine Daten werden in einer Amazon DynamoDB DynamoDB-Datenbank gespeichert. Ein Lambda-Serviceendpunkt und eine Standardroute werden zu Refactor Spaces hinzugefügt, und API Gateway wird automatisch so konfiguriert, dass die Aufrufe an die Lambda-Funktion weitergeleitet werden. Einzelheiten zur Implementierung finden Sie in Modul 2 im Iterative App Modernization Workshop.

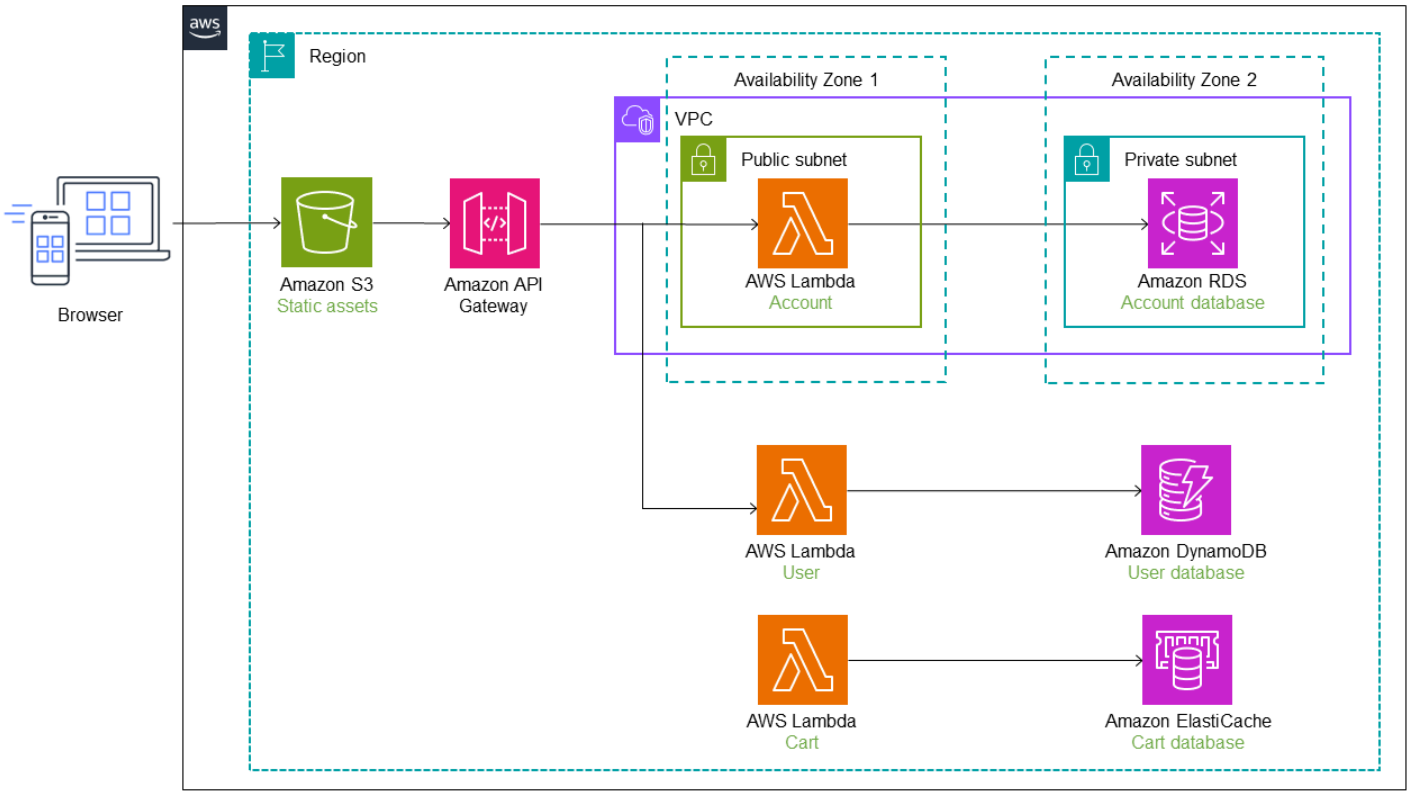


In der folgenden Abbildung wurde der Warenkorb-Service ebenfalls aus dem Monolith in eine Lambda-Funktion migriert. Eine zusätzliche Route und ein zusätzlicher Service-Endpunkt werden zu Refactor Spaces hinzugefügt, und der Datenverkehr wird automatisch auf die `Cart` Lambda-Funktion übertragen. Der Datenspeicher für die Lambda-Funktion wird von Amazon ElastiCache verwaltet.

Die monolithische Anwendung verbleibt zusammen mit der Amazon RDS-Datenbank weiterhin in der EC2-Instance.



Im nächsten Diagramm wird der letzte Dienst (Konto) aus dem Monolith in eine Lambda-Funktion migriert. Es verwendet weiterhin die ursprüngliche Amazon RDS-Datenbank. Die neue Architektur umfasst jetzt drei Microservices mit separaten Datenbanken. Jeder Dienst verwendet einen anderen Datenbanktyp. Dieses Konzept der Verwendung speziell entwickelter Datenbanken zur Erfüllung der spezifischen Anforderungen von Microservices wird als polyglotte Persistenz bezeichnet. Die Lambda-Funktionen können je nach Anwendungsfall auch in verschiedenen Programmiersprachen implementiert werden. Während des Refactorings automatisiert Refactor Spaces die Umstellung und Weiterleitung des Datenverkehrs zu Lambda. Dies spart Ihren Entwicklern die Zeit, die sie für den Entwurf, die Bereitstellung und Konfiguration der Routing-Infrastruktur benötigen.



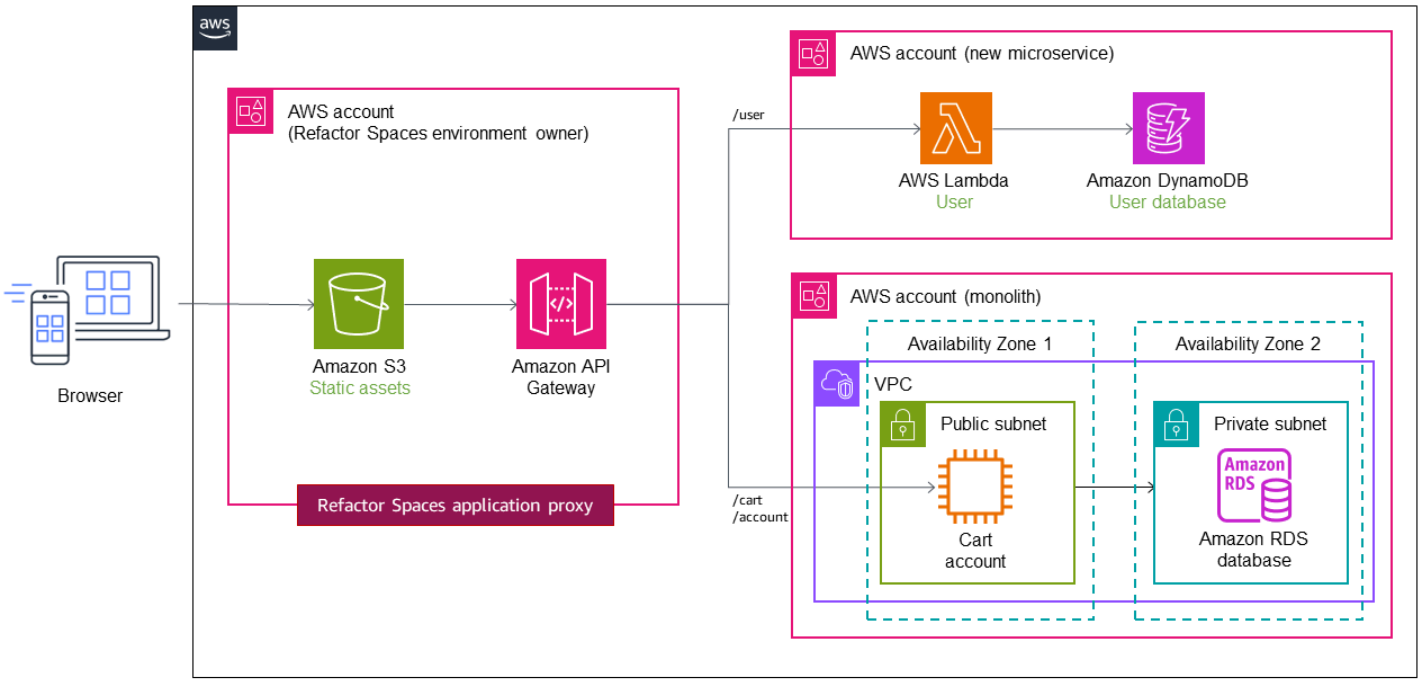
Verwenden mehrerer Konten

In der vorherigen Implementierung haben wir eine einzelne VPC mit einem privaten und einem öffentlichen Subnetz für die monolithische Anwendung verwendet und die Microservices der Einfachheit halber innerhalb derselben AWS-Konto bereitgestellt. Dies ist jedoch in realen Szenarien, in denen Microservices aus Gründen der Bereitstellungsunabhängigkeit häufig in mehreren bereitgestellt werden, selten der Fall. AWS-Konten In einer Struktur mit mehreren Konten müssen Sie die Weiterleitung des Datenverkehrs vom Monolith zu den neuen Diensten in verschiedenen Konten konfigurieren.

Refactor Spaces hilft Ihnen bei der Erstellung und Konfiguration der AWS Infrastruktur für das Routing von API-Aufrufen außerhalb der monolithischen Anwendung. Refactor Spaces orchestriert API Gateway, Network Load Balancer und ressourcenbasierte AWS Identity and Access Management(IAM) Richtlinien innerhalb Ihrer AWS Konten als Teil seiner Anwendungsressource. Sie können neue Dienste in einem einzelnen AWS-Konto oder mehreren Konten transparent zu einem externen HTTP-Endpunkt hinzufügen. All diese Ressourcen sind innerhalb Ihres orchestriert AWS-Konto und können nach der Bereitstellung angepasst und konfiguriert werden.

Gehen wir davon aus, dass die Benutzer- und Warenkorbdienste für zwei verschiedene Konten bereitgestellt werden, wie in der folgenden Abbildung dargestellt. Wenn Sie Refactor Spaces

verwenden, müssen Sie nur den Service-Endpunkt und die Route konfigurieren. Refactor Spaces automatisiert die API Gateway Gateway-Lambda-Integration und die Erstellung von Lambda-Ressourcenrichtlinien, sodass Sie sich darauf konzentrieren können, Dienste sicher außerhalb des Monolithen umzugestalten.



Ein Video-Tutorial zur Verwendung von Refactor Spaces finden Sie unter Apps inkrementell umgestalten mit. AWS Migration Hub Refactor Spaces

Workshop

- Workshop zur Iterativen Modernisierung von Apps

Blog-Referenzen

- AWS Migration Hub Refactor Spaces
- Tauchen Sie tief in ein AWS Migration Hub Refactor Spaces
- Referenzarchitektur und Referenzimplementierungen für Bereitstellungspipelines

Verwandter Inhalt

- API-Routingmuster
- Dokumentation zu Refactor Spaces

Transactional-Outbox-Muster

Absicht

Das Transactional-Outbox-Muster löst das Problem der dualen Schreiboperationen, das in verteilten Systemen auftritt, wenn eine einzelne Operation sowohl eine Datenbank-Schreiboperation als auch eine Nachrichten- oder Ereignisbenachrichtigung beinhaltet. Ein dualer Schreibvorgang tritt auf, wenn eine Anwendung in zwei verschiedene Systeme schreibt, z. B. wenn ein Microservice Daten in der Datenbank speichern und eine Nachricht senden muss, um andere Systeme zu benachrichtigen. Ein Fehler bei einem dieser Vorgänge kann zu inkonsistenten Daten führen.

Motivation

Wenn ein Microservice nach einer Datenbankaktualisierung eine Ereignisbenachrichtigung sendet, sollten diese beiden Operationen atomar ablaufen, um Datenkonsistenz und Zuverlässigkeit zu gewährleisten.

- Wenn die Datenbankaktualisierung erfolgreich ist, die Ereignisbenachrichtigung jedoch fehlschlägt, bemerkt der nachgelagerte Service die Änderung nicht und das System kann in einen inkonsistenten Zustand übergehen.
- Wenn die Datenbankaktualisierung fehlschlägt, die Ereignisbenachrichtigung jedoch gesendet wird, können Daten beschädigt werden, was die Zuverlässigkeit des Systems beeinträchtigen kann.

Anwendbarkeit

Verwenden Sie das Transactional-Outbox-Muster, wenn:

- Sie eine ereignisgesteuerte Anwendung erstellen, bei der eine Datenbankaktualisierung eine Ereignisbenachrichtigung auslöst.
- Sie sollten die Atomizität bei Vorgängen sicherstellen, an denen zwei Services beteiligt sind.
- Sie sollten das Event-Sourcing-Muster implementieren.

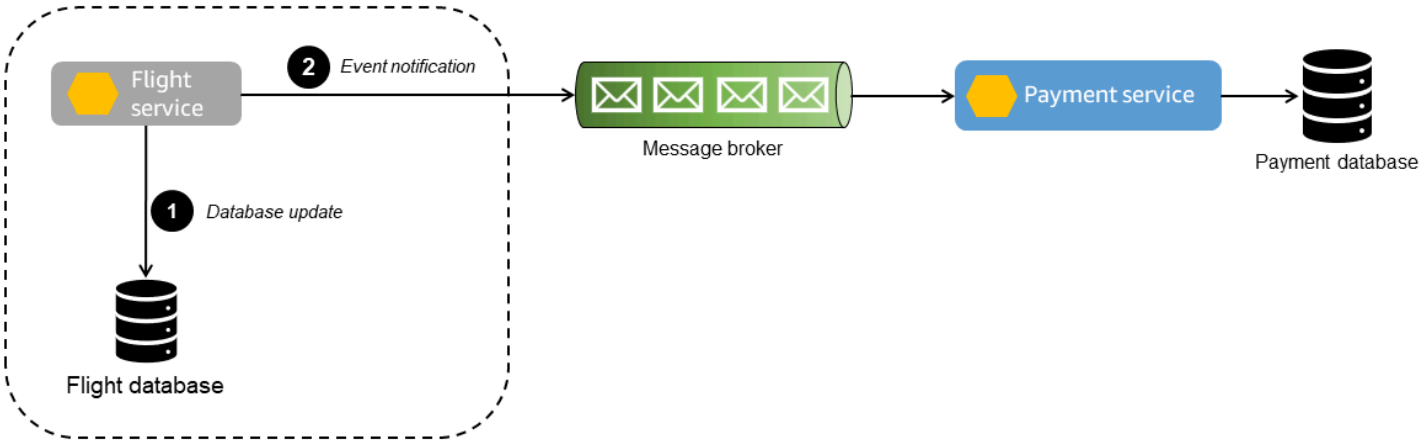
Fehler und Überlegungen

- Doppelte Nachrichten: Der Ereignisverarbeitungsservice sendet möglicherweise doppelte Nachrichten oder Ereignisse aus. Daher empfehlen wir Ihnen, den konsumierenden Service idempotent zu machen, indem Sie die verarbeiteten Nachrichten verfolgen.
- Reihenfolge der Benachrichtigungen: Senden Sie Nachrichten oder Ereignisse in derselben Reihenfolge, in der der Service die Datenbank aktualisiert. Dies ist für das Ereignis-Sourcing-Muster von entscheidender Bedeutung, bei dem Sie einen Ereignisspeicher für die point-in-time Wiederherstellung des Datenspeichers verwenden können. Wenn die Reihenfolge falsch ist, kann dies die Qualität der Daten beeinträchtigen. Ereigniskonsistenz und ein Rollback der Datenbank können das Problem noch verschärfen, wenn die Reihenfolge der Benachrichtigungen nicht beibehalten wird.
- Transaktions-Rollback: Senden Sie keine Ereignisbenachrichtigung, wenn die Transaktion rückgängig gemacht wird.
- Transaktionsverarbeitung auf Serviceebene: Wenn die Transaktion Services umfasst, für die Datenspeicher-Aktualisierungen erforderlich sind, verwenden Sie das Saga-Orchestrierungsmuster, um die Datenintegrität in allen Datenspeichern zu gewährleisten.

Implementierung

Hochrangige Architektur

Das folgende Sequenzdiagramm zeigt die Reihenfolge der Ereignisse, die bei dualen Schreibvorgängen auftreten.



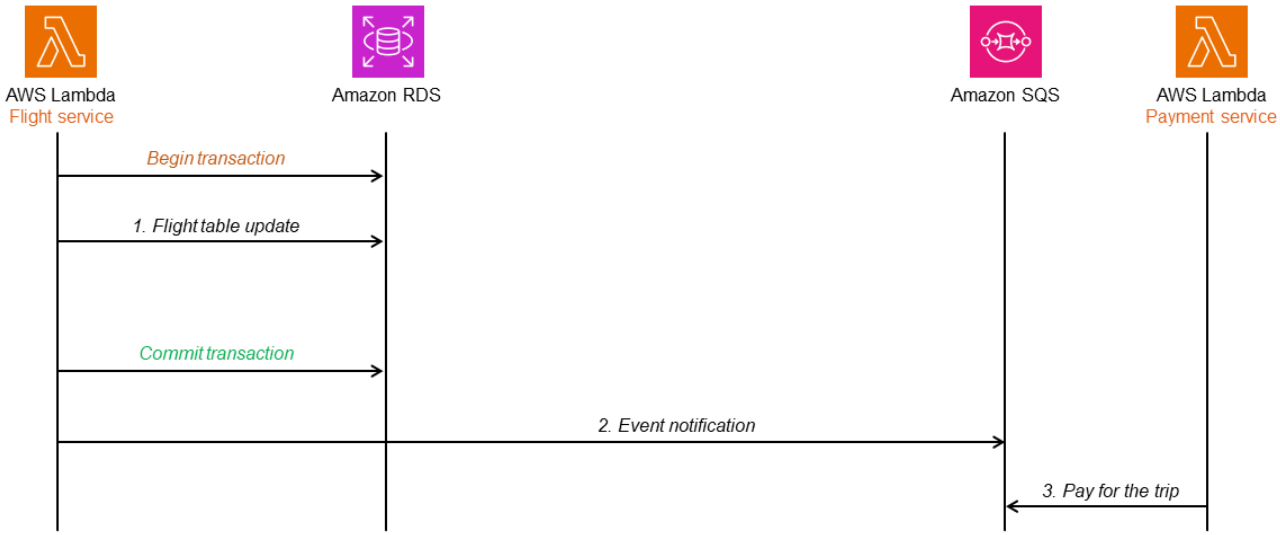
1. Der Flugservice schreibt in die Datenbank und sendet eine Ereignisbenachrichtigung an den Zahlungsservice.
2. Der Message Broker leitet die Nachrichten und Ereignisse an den Zahlungsservice weiter. Jeder Fehler im Message Broker verhindert, dass der Zahlungsservice die Updates empfängt.

Wenn die Aktualisierung der Flug-Datenbank fehlschlägt, die Benachrichtigung jedoch gesendet wird, verarbeitet der Zahlungsservice die Zahlung auf der Grundlage der Ereignisbenachrichtigung. Dies führt zu Inkonsistenzen der nachgelagerten Daten.

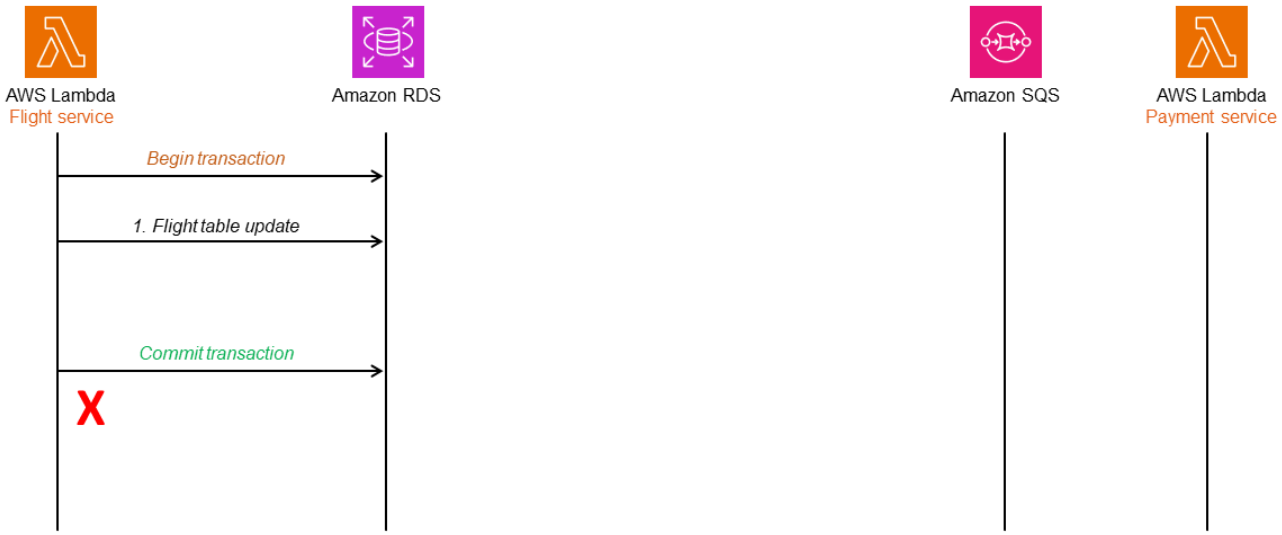
Implementierung mithilfe von AWS -Services

Um das Muster im Sequenzdiagramm zu demonstrieren, verwenden wir die folgenden AWS Services, wie im folgenden Diagramm gezeigt.

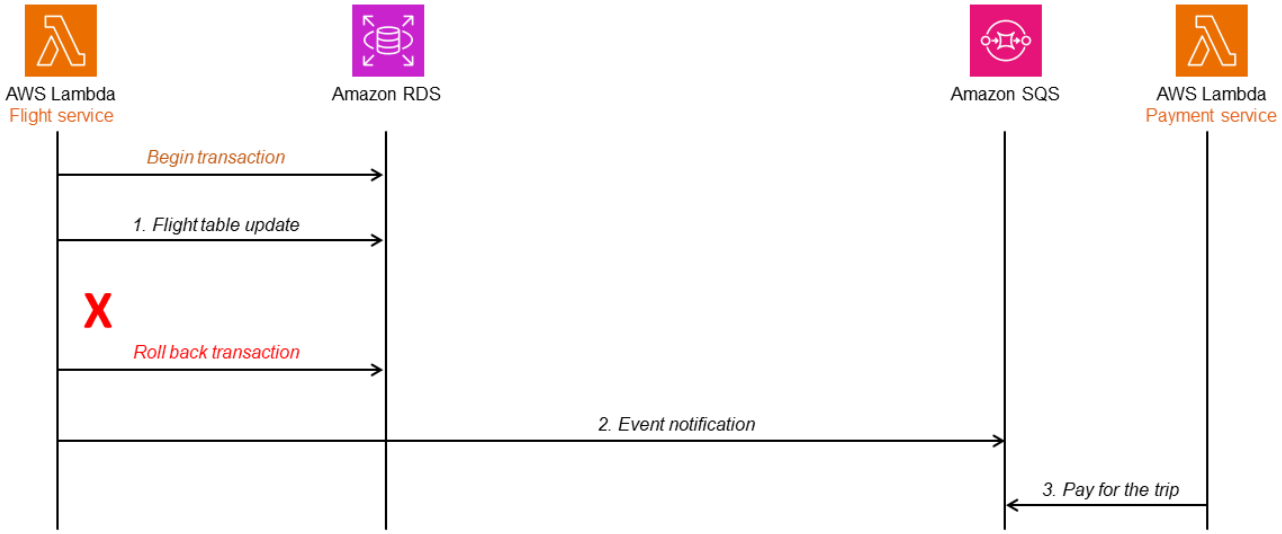
- Microservices werden durch die Verwendung von AWS Lambda implementiert.
- Die primäre Datenbank wird von Amazon Relational Database Service (Amazon RDS) verwaltet.
- Amazon Simple Queue Service (Amazon SQS) fungiert als Message Broker, der Ereignisbenachrichtigungen empfängt.



Wenn der Flugservice nach der Bestätigung der Transaktion fehlschlägt, kann dies dazu führen, dass die Ereignisbenachrichtigung nicht gesendet wird.



Allerdings könnte die Transaktion fehlschlagen und zurückgesetzt werden, wobei die Ereignisbenachrichtigung dennoch gesendet wird, so dass der Zahlungsservice die Zahlung abwickeln kann.



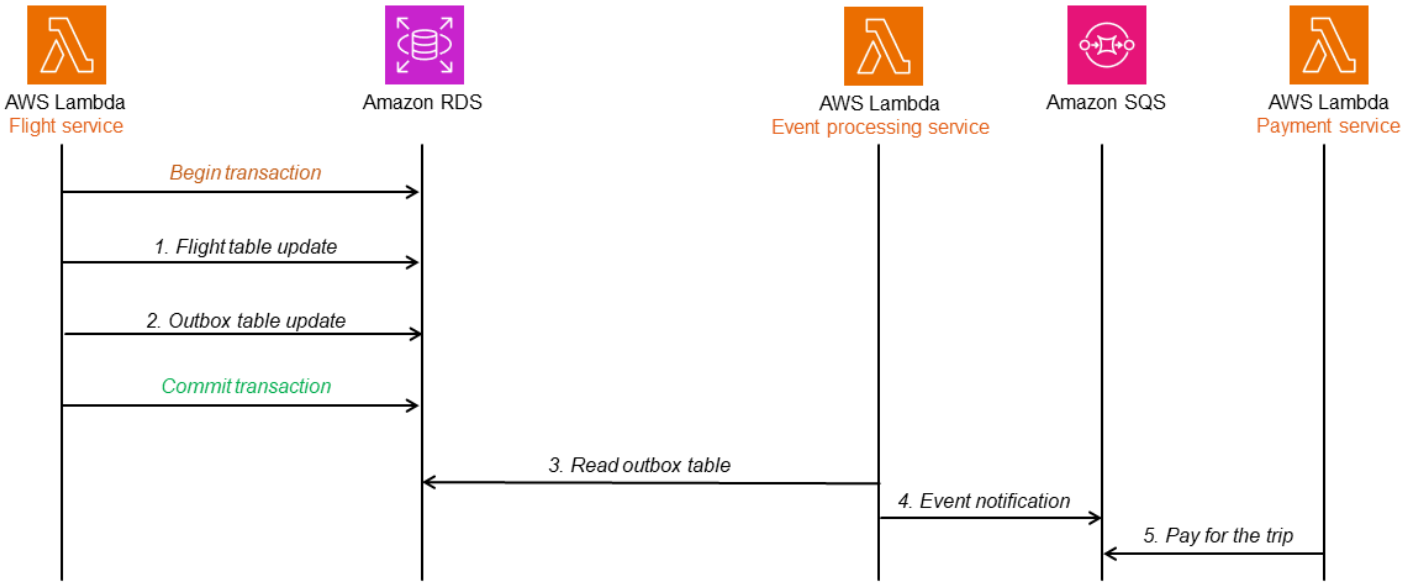
Um dieses Problem zu beheben, können Sie eine Outbox-Tabelle oder Change Data Capture (CDC) verwenden. In den folgenden Abschnitten werden diese beiden Optionen erörtert und erklärt, wie Sie diese mithilfe von AWS-Services implementieren können.

Verwenden einer Outbox-Tabelle mit einer relationalen Datenbank

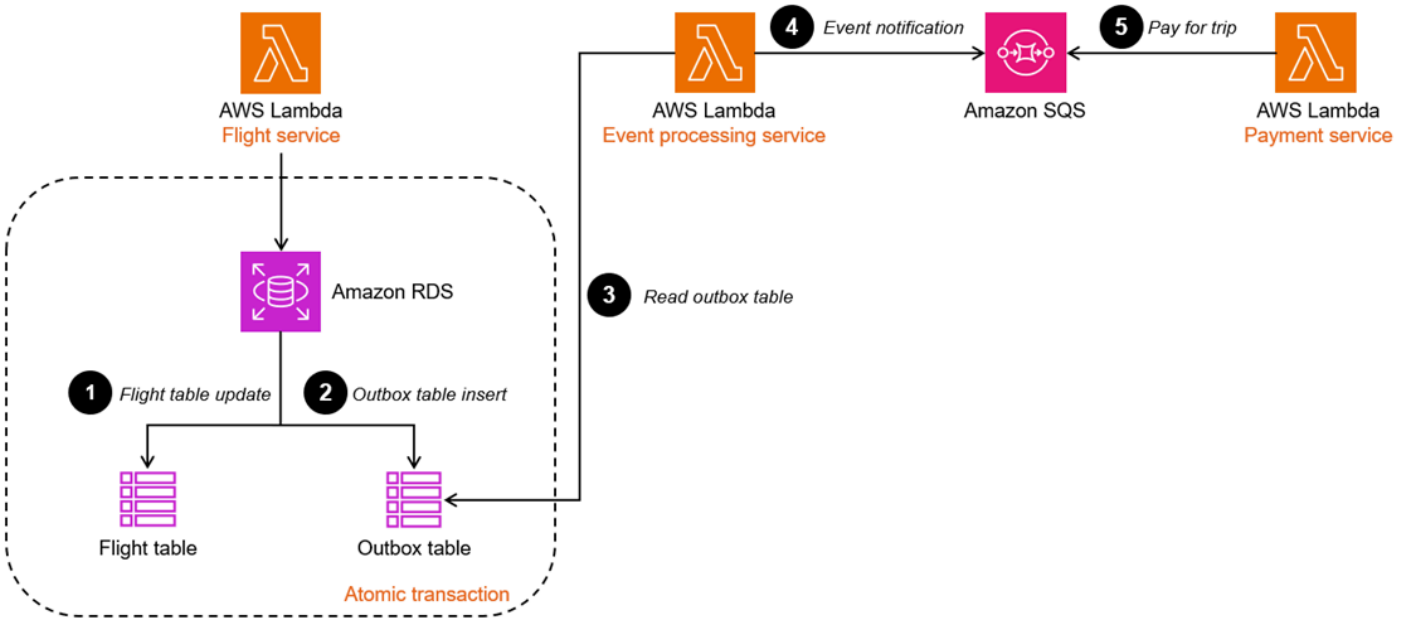
In einer Outbox-Tabelle werden alle Ereignisse des Flugservices mit einem Zeitstempel und einer Sequenznummer gespeichert.

Wenn die Flugtabelle aktualisiert wird, wird auch die Outbox-Tabelle in derselben Transaktion aktualisiert. Ein anderer Service (z. B. der Eregnisverarbeitungsservice) liest aus der Outbox-Tabelle und sendet das Ereignis an Amazon SQS. Amazon SQS sendet eine Nachricht über das Ereignis an den Zahlungsservice zur weiteren Bearbeitung. Die Amazon-SQS-Standard-Warteschlangen garantieren, dass die Nachricht mindestens einmal zugestellt wird und nicht verloren geht. Wenn Sie jedoch Amazon-SQS-Standard-Warteschlangen verwenden, wird dieselbe Nachricht oder dasselbe Ereignis möglicherweise mehr als einmal zugestellt. Daher sollten Sie die Idempotenz des Ereignisbenachrichtigungsservices sicherstellen (d. h., die mehrfache Verarbeitung derselben Nachricht sollte keine negativen Auswirkungen haben). Wenn Sie möchten, dass die Nachricht genau einmal mit der Nachrichtenreihenfolge zugestellt wird, können Sie Amazon-SQS-Warteschlangen mit FIFO (first in, first out) verwenden.

Schlägt die Aktualisierung der Flugtabelle oder die Aktualisierung der Ausgangstabelle fehl, wird die gesamte Transaktion zurückgesetzt, sodass keine Inkonsistenzen bei nachgelagerten Daten auftreten.



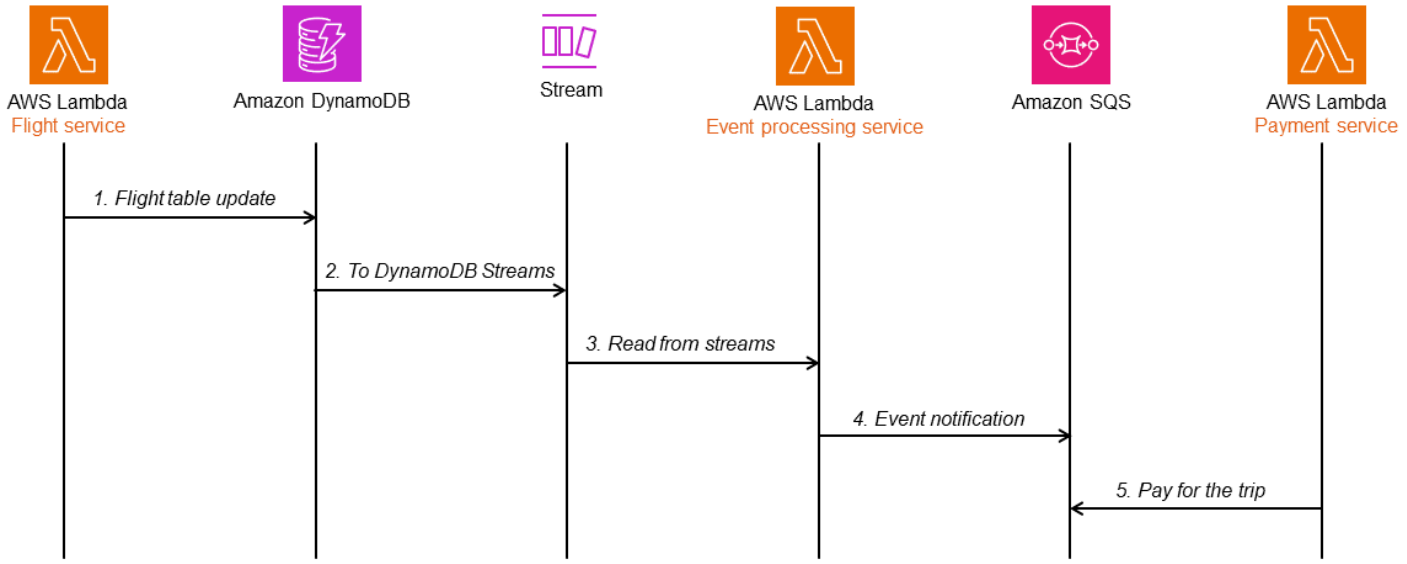
In der folgenden Abbildung wird die Transactional-Outbox-Architektur mithilfe einer Amazon RDS-Datenbank implementiert. Wenn der Ereignisverarbeitungsservice die Outbox-Tabelle liest, erkennt er nur die Zeilen, die Teil einer bestätigten (erfolgreichen) Transaktion sind, und stellt dann die Nachricht für das Ereignis in die SQS-Warteschlange, die vom Zahlungsservice zur weiteren Verarbeitung gelesen wird. Dieses Design löst das Problem der doppelten Schreiboperationen und bewahrt die Reihenfolge der Nachrichten und Ereignisse durch die Verwendung von Zeitstempeln und Sequenznummern.



Verwenden von Change Data Capture (CDC)

Einige Datenbanken unterstützen die Veröffentlichung von Änderungen auf Elementebene, um geänderte Daten zu erfassen. Sie können die geänderten Elemente identifizieren und eine entsprechende Ereignisbenachrichtigung senden. Dies erspart den Aufwand, eine weitere Tabelle zur Nachverfolgung der Aktualisierungen zu erstellen. Das vom Flugservice ausgelöste Ereignis wird in einem anderen Attribut desselben Elements gespeichert.

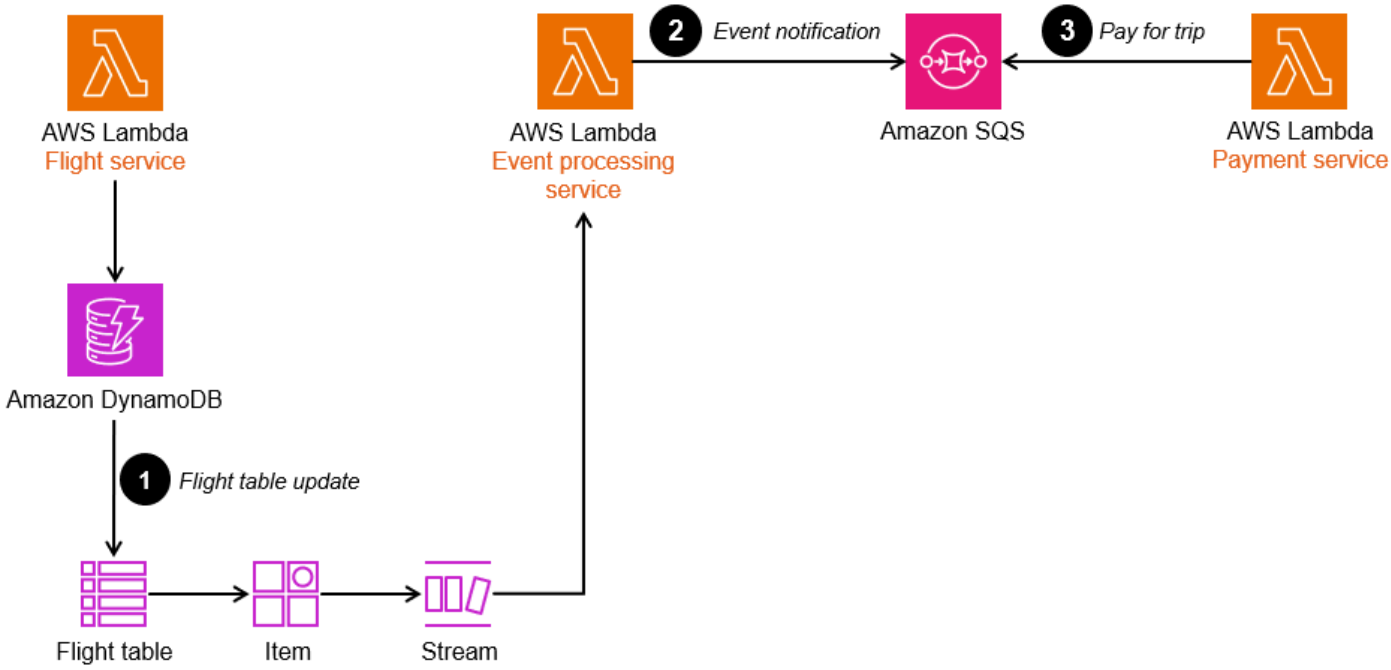
Amazon DynamoDB ist eine NoSQL-Schlüsselwert-Datenbank, die CDC-Updates unterstützt. Im folgenden Sequenzdiagramm veröffentlicht DynamoDB Änderungen auf Elementebene an Amazon DynamoDB Streams. Der Ereignisverarbeitungsservice liest aus den Streams und veröffentlicht die Ereignismeldung an den Zahlungsservice zur weiteren Verarbeitung.



DynamoDB Streams erfasst den Informationsfluss in Bezug auf Änderungen auf Elementebene in einer DynamoDB-Tabelle mithilfe einer zeitlich geordneten Reihenfolge.

Sie können ein Transactional-Outbox-Muster implementieren, indem Sie Streams in der DynamoDB-Tabelle aktivieren. Die Lambda-Funktion für den Ereignisverarbeitungsservice ist diesen Streams zugeordnet.

- Wenn die Flugtabelle aktualisiert wird, werden die geänderten Daten von DynamoDB Streams erfasst, und der Ereignisverarbeitungsservice fragt den Stream nach neuen Datensätzen ab.
- Wenn neue Stream-Datensätze verfügbar werden, platziert die Lambda-Funktion die Nachricht für das Ereignis synchron zur weiteren Verarbeitung in die SQS-Warteschlange. Sie können dem DynamoDB-Element ein Attribut hinzufügen, um den Zeitstempel und die Sequenznummer nach Bedarf zu erfassen, um die Robustheit der Implementierung zu verbessern.



Beispiel-Code

Verwenden einer Outbox-Tabelle

Der Beispielcode in diesem Abschnitt zeigt, wie Sie das transaktionale Outbox-Muster mithilfe einer Outbox-Tabelle implementieren können. Den vollständigen Code finden Sie im GitHubRepository für dieses Beispiel.

Der folgende Codeausschnitt speichert die `Flight`-Entität und das `Flight`-Ereignis in der Datenbank in ihren jeweiligen Tabellen innerhalb einer einzigen Transaktion.

```
@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;
    }
```

Ein separater Service ist dafür zuständig, die Outbox-Tabelle regelmäßig nach neuen Ereignissen zu durchsuchen, sie an Amazon SQS zu senden und sie aus der Tabelle zu löschen, wenn Amazon SQS erfolgreich antwortet. Die Abfragerate ist in der `application.properties`-Datei konfigurierbar.

```
@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);
        }
    }
```

Verwenden von Change Data Capture (CDC)

Der Beispielcode in diesem Abschnitt zeigt, wie Sie das transaktionale Outbox-Muster mithilfe der CDC-Funktionen (Change Data Capture) von DynamoDB implementieren können. Den vollständigen Code finden Sie im GitHubRepository für dieses Beispiel.

Der folgende AWS Cloud Development Kit (AWS CDK) Codeausschnitt erstellt eine DynamoDB-Flottentabelle und einen Amazon Kinesis Data Stream (`cdcStream`) und konfiguriert die Flugtabelle so, dass alle ihre Aktualisierungen an den Stream gesendet werden.

```
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,
    }

});
```

Der folgende Codeausschnitt und die folgende Konfiguration definieren eine Funktion für den Spring-Cloud-Stream, die die Updates im Kinesis-Stream aufnimmt und diese Ereignisse zur weiteren Verarbeitung an eine SQS-Warteschlange weiterleitet.

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

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

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

```
        logger.error("Error sending message to SQS", e);
    }
}
```

GitHub -Repository

Eine vollständige Implementierung der Beispielarchitektur für dieses Muster finden Sie im GitHub Repository unter https://github.com/aws-samples/transactional-outbox-pattern.

Ressourcen

Verweise

- AWSZentrum für Architektur
- AWS-Entwicklerzentrum
- Die Amazon Builders-Bibliothek

Tools

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

Methodologien

- Die Zwölf-Faktor-App(ePub von Adam Wiggins)
- Nygård, Michael T.Lass es los! : Entwicklung und Bereitstellung produktionsreifer Software. 2. Aufl. Raleigh, NC: Pragmatisches Bücherregal, 2018.
- Polyglotte Persistenz(Blogbeitrag von Martin Fowler)
- StranglerFigApplication(Blogbeitrag von Martin Fowler)

Dokumentverlauf

In der folgenden Tabelle werden wichtige Änderungen in diesem Leitfaden beschrieben. Um Benachrichtigungen über zukünftige Aktualisierungen zu erhalten, können Sie einen RSS-Feed abonnieren.

Änderung	Beschreibung	Datum
Neue Muster	Zwei neue Muster wurden hinzugefügt: Hexagonal Architecture und Scatter-Gather.	7. Mai 2024
Neue Codebeispiele	Beispielcode für den Anwendungsfall Change Data Capture (CDC) wurde zum Muster für den Transaktionsausgang hinzugefügt.	23. Februar 2024
Neue Codebeispiele	• Das Muster für den Transaktionsausgang wurde mit Beispielcode aktualisiert. • Der Abschnitt über Orchestrierung und Choreographie-Muster wurde entfernt, da diese durch Saga-Choreographie und Saga-Orchestrierung ersetzt wurden.	16. November 2023
Neue Muster	Drei neue Muster wurden hinzugefügt: Saga-Choreographie, Publish-Subscribe und Ereignis-Sourcing.	14. November 2023

Aktualisieren	Der Abschnitt zur Implementierung des Strangler-Fig-Musters wurde aktualisiert.	2. Oktober 2023
Erste Veröffentlichung	Diese erste Version enthält acht Designmuster: Anti-Corruption Layer (ACL), API-Routing, Circuit Breaker, Orchestrierung und Choreographie, Wiederholungsversuch mit Backoff, Saga-Orchestrierung, Strangler Fig und Transaktionsausgang.	28. Juli 2023

AWS Glossar zu präskriptiven Leitlinien

Im Folgenden finden Sie häufig verwendete Begriffe in Strategien, Leitfäden und Mustern von AWS Prescriptive Guidance. Um Einträge vorzuschlagen, verwenden Sie bitte den Link Feedback geben am Ende des Glossars.

Zahlen

7 Rs

Sieben gängige Migrationsstrategien für die Verlagerung von Anwendungen in die Cloud. Diese Strategien bauen auf den 5 Rs auf, die Gartner 2011 identifiziert hat, und bestehen aus folgenden Elementen:

- Faktorwechsel/Architekturwechsel – Verschieben Sie eine Anwendung und ändern Sie ihre Architektur, indem Sie alle Vorteile cloudnativer Feature nutzen, um Agilität, Leistung und Skalierbarkeit zu verbessern. Dies beinhaltet in der Regel die Portierung des Betriebssystems und der Datenbank. Beispiel: Migrieren Sie Ihre lokale Oracle-Datenbank auf die Amazon Aurora PostgreSQL-kompatible Edition.
- Plattformwechsel (Lift and Reshape) – Verschieben Sie eine Anwendung in die Cloud und führen Sie ein gewisses Maß an Optimierung ein, um die Cloud-Funktionen zu nutzen. Beispiel: Migrieren Sie Ihre lokale Oracle-Datenbank zu Amazon Relational Database Service (Amazon RDS) für Oracle in der. AWS Cloud
- Neukauf (Drop and Shop) – Wechseln Sie zu einem anderen Produkt, indem Sie typischerweise von einer herkömmlichen Lizenz zu einem SaaS-Modell wechseln. Beispiel: Migrieren Sie Ihr CRM-System (Customer Relationship Management) zu Salesforce.com.
- Hostwechsel (Lift and Shift) – Verschieben Sie eine Anwendung in die Cloud, ohne Änderungen vorzunehmen, um die Cloud-Funktionen zu nutzen. Beispiel: Migrieren Sie Ihre lokale Oracle-Datenbank zu Oracle auf einer EC2-Instanz in der. AWS Cloud
- Verschieben (Lift and Shift auf Hypervisor-Ebene) – Verlagern Sie die Infrastruktur in die Cloud, ohne neue Hardware kaufen, Anwendungen umschreiben oder Ihre bestehenden Abläufe ändern zu müssen. Sie migrieren Server von einer lokalen Plattform zu einem Cloud-Dienst für dieselbe Plattform. Beispiel: Migrieren Sie eine Microsoft Hyper-V Anwendung zu AWS.
- Beibehaltung (Wiederaufgreifen) – Bewahren Sie Anwendungen in Ihrer Quellumgebung auf. Dazu können Anwendungen gehören, die einen umfangreichen Faktorwechsel erfordern und

die Sie auf einen späteren Zeitpunkt verschieben möchten, sowie ältere Anwendungen, die Sie beibehalten möchten, da es keine geschäftliche Rechtfertigung für ihre Migration gibt.

- Außerbetriebnahme – Dekommissionierung oder Entfernung von Anwendungen, die in Ihrer Quellumgebung nicht mehr benötigt werden.

A

ABAC

Siehe attributbasierte Zugriffskontrolle.

abstrahierte Dienste

Weitere Informationen finden Sie unter Managed Services.

ACID

Siehe Atomarität, Konsistenz, Isolierung und Haltbarkeit.

Aktiv-Aktiv-Migration

Eine Datenbankmigrationsmethode, bei der die Quell- und Zieldatenbanken synchron gehalten werden (mithilfe eines bidirektionalen Replikationstools oder dualer Schreibvorgänge) und beide Datenbanken Transaktionen von miteinander verbundenen Anwendungen während der Migration verarbeiten. Diese Methode unterstützt die Migration in kleinen, kontrollierten Batches, anstatt einen einmaligen Cutover zu erfordern. Es ist flexibler, erfordert aber mehr Arbeit als eine aktiv-passive Migration.

Aktiv-Passiv-Migration

Eine Datenbankmigrationsmethode, bei der die Quell- und Zieldatenbanken synchron gehalten werden, aber nur die Quelldatenbank Transaktionen von verbindenden Anwendungen verarbeitet, während Daten in die Zieldatenbank repliziert werden. Die Zieldatenbank akzeptiert während der Migration keine Transaktionen.

Aggregatfunktion

Eine SQL-Funktion, die mit einer Gruppe von Zeilen arbeitet und einen einzelnen Rückgabewert für die Gruppe berechnet. Beispiele für Aggregatfunktionen sind `SUM` und`MAX`.

AI

Siehe künstliche Intelligenz.

AIOps

Siehe Operationen mit künstlicher Intelligenz.

Anonymisierung

Der Prozess des dauerhaften Löschens personenbezogener Daten in einem Datensatz. Anonymisierung kann zum Schutz der Privatsphäre beitragen. Anonymisierte Daten gelten nicht mehr als personenbezogene Daten.

Anti-Muster

Eine häufig verwendete Lösung für ein wiederkehrendes Problem, bei dem die Lösung kontraproduktiv, ineffektiv oder weniger wirksam als eine Alternative ist.

Anwendungssteuerung

Ein Sicherheitsansatz, bei dem nur zugelassene Anwendungen verwendet werden können, um ein System vor Schadsoftware zu schützen.

Anwendungsportfolio

Eine Sammlung detaillierter Informationen zu jeder Anwendung, die von einer Organisation verwendet wird, einschließlich der Kosten für die Erstellung und Wartung der Anwendung und ihres Geschäftswerts. Diese Informationen sind entscheidend für den Prozess der Portfoliofindung und -analyse und hilft bei der Identifizierung und Priorisierung der Anwendungen, die migriert, modernisiert und optimiert werden sollen.

künstliche Intelligenz (KI)

Das Gebiet der Datenverarbeitungswissenschaft, das sich der Nutzung von Computertechnologien zur Ausführung kognitiver Funktionen widmet, die typischerweise mit Menschen in Verbindung gebracht werden, wie Lernen, Problemlösen und Erkennen von Mustern. Weitere Informationen finden Sie unter Was ist künstliche Intelligenz?

Operationen mit künstlicher Intelligenz (AIOps)

Der Prozess des Einsatzes von Techniken des Machine Learning zur Lösung betrieblicher Probleme, zur Reduzierung betrieblicher Zwischenfälle und menschlicher Eingriffe sowie zur Steigerung der Servicequalität. Weitere Informationen zur Verwendung von AIOps in der AWS -Migrationsstrategie finden Sie im Leitfaden zur Betriebsintegration.

Asymmetrische Verschlüsselung

Ein Verschlüsselungsalgorithmus, der ein Schlüsselpaar, einen öffentlichen Schlüssel für die Verschlüsselung und einen privaten Schlüssel für die Entschlüsselung verwendet. Sie können den

öffentlichen Schlüssel teilen, da er nicht für die Entschlüsselung verwendet wird. Der Zugriff auf den privaten Schlüssel sollte jedoch stark eingeschränkt sein.

Atomizität, Konsistenz, Isolierung, Haltbarkeit (ACID)

Eine Reihe von Softwareeigenschaften, die die Datenvalidität und betriebliche Zuverlässigkeit einer Datenbank auch bei Fehlern, Stromausfällen oder anderen Problemen gewährleisten.

Attributbasierte Zugriffskontrolle (ABAC)

Die Praxis, detaillierte Berechtigungen auf der Grundlage von Benutzerattributen wie Abteilung, Aufgabenrolle und Teamname zu erstellen. Weitere Informationen finden Sie unter ABAC AWS in der AWS Identity and Access Management (IAM-) Dokumentation.

autoritative Datenquelle

Ein Ort, an dem Sie die primäre Version der Daten speichern, die als die zuverlässigste Informationsquelle angesehen wird. Sie können Daten aus der maßgeblichen Datenquelle an andere Speicherorte kopieren, um die Daten zu verarbeiten oder zu ändern, z. B. zu anonymisieren, zu redigieren oder zu pseudonymisieren.

Availability Zone

Ein bestimmter Standort innerhalb einer AWS-Region , der vor Ausfällen in anderen Availability Zones geschützt ist und kostengünstige Netzwerkkonnektivität mit niedriger Latenz zu anderen Availability Zones in derselben Region bietet.

AWS Framework für die Cloud-Einführung (AWS CAF)

Ein Framework mit Richtlinien und bewährten Verfahren, das Unternehmen bei der Entwicklung eines effizienten und effektiven Plans für den erfolgreichen Umstieg auf die Cloud unterstützt. AWS AWS CAF unterteilt die Leitlinien in sechs Schwerpunktbereiche, die als Perspektiven bezeichnet werden: Unternehmen, Mitarbeiter, Unternehmensführung, Plattform, Sicherheit und Betrieb. Die Perspektiven Geschäft, Mitarbeiter und Unternehmensführung konzentrieren sich auf Geschäftskompetenzen und -prozesse, während sich die Perspektiven Plattform, Sicherheit und Betriebsabläufe auf technische Fähigkeiten und Prozesse konzentrieren. Die Personalperspektive zielt beispielsweise auf Stakeholder ab, die sich mit Personalwesen (HR), Personalfunktionen und Personalmanagement befassen. Aus dieser Perspektive bietet AWS CAF Leitlinien für Personalentwicklung, Schulung und Kommunikation, um das Unternehmen auf eine erfolgreiche Cloud-Einführung vorzubereiten. Weitere Informationen finden Sie auf der AWS -CAF-Webseite und dem AWS -CAF-Whitepaper.

AWS Workload-Qualifizierungsrahmen (AWS WQF)

Ein Tool, das Workloads bei der Datenbankmigration bewertet, Migrationsstrategien empfiehlt und Arbeitsschätzungen bereitstellt. AWS WQF ist in () enthalten. AWS Schema Conversion Tool AWS SCT Es analysiert Datenbankschemas und Codeobjekte, Anwendungscode, Abhängigkeiten und Leistungsmerkmale und stellt Bewertungsberichte bereit.

B

schlechter Bot

Ein Bot, der Einzelpersonen oder Organisationen stören oder ihnen Schaden zufügen soll.

BCP

Siehe Planung der Geschäftskontinuität.

Verhaltensdiagramm

Eine einheitliche, interaktive Ansicht des Ressourcenverhaltens und der Interaktionen im Laufe der Zeit. Sie können ein Verhaltensdiagramm mit Amazon Detective verwenden, um fehlgeschlagene Anmeldeversuche, verdächtige API-Aufrufe und ähnliche Vorgänge zu untersuchen. Weitere Informationen finden Sie unter Daten in einem Verhaltensdiagramm in der Detective-Dokumentation.

Big-Endian-System

Ein System, welches das höchstwertige Byte zuerst speichert. Siehe auch Endianness.

Binäre Klassifikation

Ein Prozess, der ein binäres Ergebnis vorhersagt (eine von zwei möglichen Klassen). Beispielsweise könnte Ihr ML-Modell möglicherweise Probleme wie „Handelt es sich bei dieser E-Mail um Spam oder nicht?“ vorhersagen müssen oder „Ist dieses Produkt ein Buch oder ein Auto?“

Bloom-Filter

Eine probabilistische, speichereffiziente Datenstruktur, mit der getestet wird, ob ein Element Teil einer Menge ist.

Blau/Grün-Bereitstellung

Eine Bereitstellungsstrategie, bei der Sie zwei separate, aber identische Umgebungen erstellen. Sie führen die aktuelle Anwendungsversion in einer Umgebung (blau) und die neue

Anwendungsversion in der anderen Umgebung (grün) aus. Mit dieser Strategie können Sie schnell und mit minimalen Auswirkungen ein Rollback durchführen.

Bot

Eine Softwareanwendung, die automatisierte Aufgaben über das Internet ausführt und menschliche Aktivitäten oder Interaktionen simuliert. Manche Bots sind nützlich oder nützlich, wie z. B. Webcrawler, die Informationen im Internet indexieren. Einige andere Bots, die als bösartige Bots bezeichnet werden, sollen Einzelpersonen oder Organisationen stören oder ihnen Schaden zufügen.

Botnetz

Netzwerke von Bots, die mit Malware infiziert sind und unter der Kontrolle einer einzigen Partei stehen, die als Bot-Herder oder Bot-Operator bezeichnet wird. Botnetze sind der bekannteste Mechanismus zur Skalierung von Bots und ihrer Wirkung.

branch

Ein containerisierter Bereich eines Code-Repositorys. Der erste Zweig, der in einem Repository erstellt wurde, ist der Hauptzweig. Sie können einen neuen Zweig aus einem vorhandenen Zweig erstellen und dann Feature entwickeln oder Fehler in dem neuen Zweig beheben. Ein Zweig, den Sie erstellen, um ein Feature zu erstellen, wird allgemein als Feature-Zweig bezeichnet. Wenn das Feature zur Veröffentlichung bereit ist, führen Sie den Feature-Zweig wieder mit dem Hauptzweig zusammen. Weitere Informationen finden Sie unter Über Branches (GitHub Dokumentation).

Zugang durch Glasbruch

Unter außergewöhnlichen Umständen und im Rahmen eines genehmigten Verfahrens ist dies eine schnelle Methode für einen Benutzer, auf einen Bereich zuzugreifen AWS-Konto , für den er normalerweise keine Zugriffsrechte besitzt. Weitere Informationen finden Sie unter dem Indikator Implementation break-glass procedures in den AWS Well-Architected-Leitlinien.

Brownfield-Strategie

Die bestehende Infrastruktur in Ihrer Umgebung. Wenn Sie eine Brownfield-Strategie für eine Systemarchitektur anwenden, richten Sie sich bei der Gestaltung der Architektur nach den Einschränkungen der aktuellen Systeme und Infrastruktur. Wenn Sie die bestehende Infrastruktur erweitern, könnten Sie Brownfield- und Greenfield-Strategien mischen.

Puffer-Cache

Der Speicherbereich, in dem die am häufigsten abgerufenen Daten gespeichert werden.

Geschäftsfähigkeit

Was ein Unternehmen tut, um Wert zu generieren (z. B. Vertrieb, Kundenservice oder Marketing). Microservices-Architekturen und Entwicklungsentscheidungen können von den Geschäftskapazitäten beeinflusst werden. Weitere Informationen finden Sie im Abschnitt Organisiert nach Geschäftskapazitäten des Whitepapers Ausführen von containerisierten Microservices in AWS.

Planung der Geschäftskontinuität (BCP)

Ein Plan, der die potenziellen Auswirkungen eines störenden Ereignisses, wie z. B. einer groß angelegten Migration, auf den Betrieb berücksichtigt und es einem Unternehmen ermöglicht, den Betrieb schnell wieder aufzunehmen.

C

CAF

Weitere Informationen finden Sie unter Framework für die AWS Cloud-Einführung.

Bereitstellung auf Kanaren

Die langsame und schrittweise Veröffentlichung einer Version für Endbenutzer. Wenn Sie sich sicher sind, stellen Sie die neue Version bereit und ersetzen die aktuelle Version vollständig.

CCoE

Weitere Informationen finden Sie im Cloud Center of Excellence.

CDC

Siehe Erfassung von Änderungsdaten.

Erfassung von Datenänderungen (CDC)

Der Prozess der Nachverfolgung von Änderungen an einer Datenquelle, z. B. einer Datenbanktabelle, und der Aufzeichnung von Metadaten zu der Änderung. Sie können CDC für verschiedene Zwecke verwenden, z. B. für die Prüfung oder Replikation von Änderungen in einem Zielsystem, um die Synchronisation aufrechtzuerhalten.

Chaos-Technik

Absichtliches Einführen von Ausfällen oder Störungsereignissen, um die Widerstandsfähigkeit eines Systems zu testen. Sie können AWS Fault Injection Service (AWS FIS) verwenden, um Experimente durchzuführen, die Ihre AWS Workloads stress, und deren Reaktion zu bewerten.

CI/CD

Siehe Continuous Integration und Continuous Delivery.

Klassifizierung

Ein Kategorisierungsprozess, der bei der Erstellung von Vorhersagen hilft. ML-Modelle für Klassifikationsprobleme sagen einen diskreten Wert voraus. Diskrete Werte unterscheiden sich immer voneinander. Beispielsweise muss ein Modell möglicherweise auswerten, ob auf einem Bild ein Auto zu sehen ist oder nicht.

clientseitige Verschlüsselung

Lokale Verschlüsselung von Daten, bevor das Ziel sie AWS -Service empfängt.

Cloud-Kompetenzzentrum (CCoE)

Ein multidisziplinäres Team, das die Cloud-Einführung in der gesamten Organisation vorantreibt, einschließlich der Entwicklung bewährter Cloud-Methoden, der Mobilisierung von Ressourcen, der Festlegung von Migrationszeitplänen und der Begleitung der Organisation durch groß angelegte Transformationen. Weitere Informationen finden Sie in den CCoE-Beiträgen im AWS Cloud Enterprise Strategy Blog.

Cloud Computing

Die Cloud-Technologie, die typischerweise für die Ferndatenspeicherung und das IoT-Gerätemanagement verwendet wird. Cloud Computing ist häufig mit Edge-Computing-Technologie verbunden.

Cloud-Betriebsmodell

In einer IT-Organisation das Betriebsmodell, das zum Aufbau, zur Weiterentwicklung und Optimierung einer oder mehrerer Cloud-Umgebungen verwendet wird. Weitere Informationen finden Sie unter Aufbau Ihres Cloud-Betriebsmodells.

Phasen der Einführung der Cloud

Die vier Phasen, die Unternehmen bei der Migration in der Regel durchlaufen AWS Cloud:

- Projekt – Durchführung einiger Cloud-bezogener Projekte zu Machbarkeitsnachweisen und zu Lernzwecken
- Fundament – Grundlegende Investitionen tätigen, um Ihre Cloud-Einführung zu skalieren (z. B. Einrichtung einer Landing Zone, Definition eines CCoE, Einrichtung eines Betriebsmodells)
- Migration – Migrieren einzelner Anwendungen

- Neuentwicklung – Optimierung von Produkten und Services und Innovation in der Cloud

Diese Phasen wurden von Stephen Orban im Blogbeitrag The Journey Toward Cloud-First & the Stages of Adoption im AWS Cloud Enterprise Strategy-Blog definiert. Informationen darüber, wie sie mit der AWS Migrationsstrategie zusammenhängen, finden Sie im Leitfaden zur Vorbereitung der Migration.

CMDB

Siehe Datenbank für das Konfigurationsmanagement.

Code-Repository

Ein Ort, an dem Quellcode und andere Komponenten wie Dokumentation, Beispiele und Skripts gespeichert und im Rahmen von Versionskontrollprozessen aktualisiert werden. Zu den gängigen Cloud-Repositorys gehören GitHub oder AWS CodeCommit. Jede Version des Codes wird Zweig genannt. In einer Microservice-Struktur ist jedes Repository einer einzelnen Funktionalität gewidmet. Eine einzelne CI/CD-Pipeline kann mehrere Repositorien verwenden.

Kalter Cache

Ein Puffer-Cache, der leer oder nicht gut gefüllt ist oder veraltete oder irrelevante Daten enthält. Dies beeinträchtigt die Leistung, da die Datenbank-Instance aus dem Hauptspeicher oder der Festplatte lesen muss, was langsamer ist als das Lesen aus dem Puffercache.

Kalte Daten

Daten, auf die selten zugegriffen wird und die in der Regel historisch sind. Bei der Abfrage dieser Art von Daten sind langsame Abfragen in der Regel akzeptabel. Durch die Verlagerung dieser Daten auf leistungsschwächere und kostengünstigere Speicherstufen oder -klassen können Kosten gesenkt werden.

Computer Vision (CV)

Ein Bereich der KI, der maschinelles Lernen nutzt, um Informationen aus visuellen Formaten wie digitalen Bildern und Videos zu analysieren und zu extrahieren. AWS Panorama Bietet beispielsweise Geräte an, die CV zu lokalen Kameranetzwerken hinzufügen, und Amazon SageMaker stellt Bildverarbeitungsalgorithmen für CV bereit.

Drift in der Konfiguration

Bei einer Arbeitslast eine Änderung der Konfiguration gegenüber dem erwarteten Zustand. Dies kann dazu führen, dass der Workload nicht mehr richtlinienkonform wird, und zwar in der Regel schrittweise und unbeabsichtigt.

Verwaltung der Datenbankkonfiguration (CMDB)

Ein Repository, das Informationen über eine Datenbank und ihre IT-Umgebung speichert und verwaltet, inklusive Hardware- und Softwarekomponenten und deren Konfigurationen. In der Regel verwenden Sie Daten aus einer CMDB in der Phase der Portfolioerkennung und -analyse der Migration.

Konformitätspaket

Eine Sammlung von AWS Config Regeln und Abhilfemaßnahmen, die Sie zusammenstellen können, um Ihre Konformitäts- und Sicherheitsprüfungen individuell anzupassen. Mithilfe einer YAML-Vorlage können Sie ein Conformance Pack als einzelne Entität in einer AWS-Konto AND-Region oder unternehmensweit bereitstellen. Weitere Informationen finden Sie in der Dokumentation unter Conformance Packs. AWS Config

Kontinuierliche Bereitstellung und kontinuierliche Integration (CI/CD)

Der Prozess der Automatisierung der Quell-, Build-, Test-, Staging- und Produktionsphasen des Softwareveröffentlichungsprozesses. CI/CD wird allgemein als Pipeline beschrieben. CI/CD kann Ihnen helfen, Prozesse zu automatisieren, die Produktivität zu steigern, die Codequalität zu verbessern und schneller zu liefern. Weitere Informationen finden Sie unter Vorteile der kontinuierlichen Auslieferung. CD kann auch für kontinuierliche Bereitstellung stehen. Weitere Informationen finden Sie unterKontinuierliche Auslieferung im Vergleich zu kontinuierlicher Bereitstellung.

CV

Siehe Computer Vision.

D

Daten im Ruhezustand

Daten, die in Ihrem Netzwerk stationär sind, z. B. Daten, die sich im Speicher befinden.

Datenklassifizierung

Ein Prozess zur Identifizierung und Kategorisierung der Daten in Ihrem Netzwerk auf der Grundlage ihrer Kritikalität und Sensitivität. Sie ist eine wichtige Komponente jeder Strategie für das Management von Cybersecurity-Risiken, da sie Ihnen hilft, die geeigneten Schutz- und Aufbewahrungskontrollen für die Daten zu bestimmen. Die Datenklassifizierung ist ein Bestandteil

der Sicherheitssäule im AWS Well-Architected Framework. Weitere Informationen finden Sie unter Datenklassifizierung.

Datendrift

Eine signifikante Abweichung zwischen den Produktionsdaten und den Daten, die zum Trainieren eines ML-Modells verwendet wurden, oder eine signifikante Änderung der Eingabedaten im Laufe der Zeit. Datendrift kann die Gesamtqualität, Genauigkeit und Fairness von ML-Modellvorhersagen beeinträchtigen.

Daten während der Übertragung

Daten, die sich aktiv durch Ihr Netzwerk bewegen, z. B. zwischen Netzwerkressourcen.

Datennetz

Ein architektonisches Framework, das verteilte, dezentrale Dateneigentum mit zentraler Verwaltung und Steuerung ermöglicht.

Datenminimierung

Das Prinzip, nur die Daten zu sammeln und zu verarbeiten, die unbedingt erforderlich sind. Durch Datenminimierung im AWS Cloud können Datenschutzrisiken, Kosten und der CO2-Fußabdruck Ihrer Analysen reduziert werden.

Datenperimeter

Eine Reihe präventiver Schutzmaßnahmen in Ihrer AWS Umgebung, die sicherstellen, dass nur vertrauenswürdige Identitäten auf vertrauenswürdige Ressourcen von erwarteten Netzwerken zugreifen. Weitere Informationen finden Sie unter Aufbau eines Datenperimeters auf. AWS

Vorverarbeitung der Daten

Rohdaten in ein Format umzuwandeln, das von Ihrem ML-Modell problemlos verarbeitet werden kann. Die Vorverarbeitung von Daten kann bedeuten, dass bestimmte Spalten oder Zeilen entfernt und fehlende, inkonsistente oder doppelte Werte behoben werden.

Herkunft der Daten

Der Prozess der Nachverfolgung des Ursprungs und der Geschichte von Daten während ihres gesamten Lebenszyklus, z. B. wie die Daten generiert, übertragen und gespeichert wurden.

betroffene Person

Eine Person, deren Daten gesammelt und verarbeitet werden.

Data Warehouse

Ein Datenverwaltungssystem, das Business Intelligence wie Analysen unterstützt. Data Warehouses enthalten in der Regel große Mengen an historischen Daten und werden in der Regel für Abfragen und Analysen verwendet.

Datenbankdefinitionssprache (DDL)

Anweisungen oder Befehle zum Erstellen oder Ändern der Struktur von Tabellen und Objekten in einer Datenbank.

Datenbankmanipulationssprache (DML)

Anweisungen oder Befehle zum Ändern (Einfügen, Aktualisieren und Löschen) von Informationen in einer Datenbank.

DDL

Siehe Datenbankdefinitionssprache.

Deep-Ensemble

Mehrere Deep-Learning-Modelle zur Vorhersage kombinieren. Sie können Deep-Ensembles verwenden, um eine genauere Vorhersage zu erhalten oder um die Unsicherheit von Vorhersagen abzuschätzen.

Deep Learning

Ein ML-Teilbereich, der mehrere Schichten künstlicher neuronaler Netzwerke verwendet, um die Zuordnung zwischen Eingabedaten und Zielvariablen von Interesse zu ermitteln.

defense-in-depth

Ein Ansatz zur Informationssicherheit, bei dem eine Reihe von Sicherheitsmechanismen und -kontrollen sorgfältig in einem Computernetzwerk verteilt werden, um die Vertraulichkeit, Integrität und Verfügbarkeit des Netzwerks und der darin enthaltenen Daten zu schützen. Wenn Sie diese Strategie anwenden AWS, fügen Sie mehrere Steuerelemente auf verschiedenen Ebenen der AWS Organizations Struktur hinzu, um die Ressourcen zu schützen. Ein defense-in-depth Ansatz könnte beispielsweise Multi-Faktor-Authentifizierung, Netzwerksegmentierung und Verschlüsselung kombinieren.

delegierter Administrator

In AWS Organizations kann ein kompatibler Dienst ein AWS Mitgliedskonto registrieren, um die Konten der Organisation und die Berechtigungen für diesen Dienst zu verwalten. Dieses Konto

wird als delegierter Administrator für diesen Service bezeichnet. Weitere Informationen und eine Liste kompatibler Services finden Sie unter Services, die mit AWS Organizations funktionieren in der AWS Organizations -Dokumentation.

Bereitstellung

Der Prozess, bei dem eine Anwendung, neue Feature oder Codekorrekturen in der Zielumgebung verfügbar gemacht werden. Die Bereitstellung umfasst das Implementieren von Änderungen an einer Codebasis und das anschließende Erstellen und Ausführen dieser Codebasis in den Anwendungsumgebungen.

Entwicklungsumgebung

Siehe Umgebung.

Detektivische Kontrolle

Eine Sicherheitskontrolle, die darauf ausgelegt ist, ein Ereignis zu erkennen, zu protokollieren und zu warnen, nachdem ein Ereignis eingetreten ist. Diese Kontrollen stellen eine zweite Verteidigungslinie dar und warnen Sie vor Sicherheitsereignissen, bei denen die vorhandenen präventiven Kontrollen umgangen wurden. Weitere Informationen finden Sie unter Detektivische Kontrolle in Implementierung von Sicherheitskontrollen in AWS.

Abbildung des Wertstroms in der Entwicklung (DVSM)

Ein Prozess zur Identifizierung und Priorisierung von Einschränkungen, die sich negativ auf Geschwindigkeit und Qualität im Lebenszyklus der Softwareentwicklung auswirken. DVSM erweitert den Prozess der Wertstromanalyse, der ursprünglich für Lean-Manufacturing-Praktiken konzipiert wurde. Es konzentriert sich auf die Schritte und Teams, die erforderlich sind, um durch den Softwareentwicklungsprozess Mehrwert zu schaffen und zu steigern.

digitaler Zwilling

Eine virtuelle Darstellung eines realen Systems, z. B. eines Gebäudes, einer Fabrik, einer Industrieanlage oder einer Produktionslinie. Digitale Zwillinge unterstützen vorausschauende Wartung, Fernüberwachung und Produktionsoptimierung.

Maßtabelle

In einem Sternschema eine kleinere Tabelle, die Datenattribute zu quantitativen Daten in einer Faktentabelle enthält. Bei Attributen von Dimensionstabellen handelt es sich in der Regel um Textfelder oder diskrete Zahlen, die sich wie Text verhalten. Diese Attribute werden häufig zum Einschränken von Abfragen, zum Filtern und zur Kennzeichnung von Ergebnismengen verwendet.

Katastrophe

Ein Ereignis, das verhindert, dass ein Workload oder ein System seine Geschäftsziele an seinem primären Einsatzort erfüllt. Diese Ereignisse können Naturkatastrophen, technische Ausfälle oder das Ergebnis menschlichen Handelns sein, z. B. unbeabsichtigte Fehlkonfigurationen oder Malware-Angriffe.

Notfallwiederherstellung (DR)

Die Strategie und der Prozess, die Sie zur Minimierung von Ausfallzeiten und Datenverlusten aufgrund einer Katastrophe anwenden. Weitere Informationen finden Sie unter Disaster Recovery von Workloads unter AWS: Wiederherstellung in der Cloud im AWS Well-Architected Framework.

DML

Siehe Sprache zur Datenbankmanipulation.

Domainorientiertes Design

Ein Ansatz zur Entwicklung eines komplexen Softwaresystems, bei dem seine Komponenten mit sich entwickelnden Domains oder Kerngeschäftszielen verknüpft werden, denen jede Komponente dient. Dieses Konzept wurde von Eric Evans in seinem Buch Domaingesteuertes Design: Bewältigen der Komplexität im Herzen der Software (Boston: Addison-Wesley Professional, 2003) vorgestellt. Informationen darüber, wie Sie domaingesteuertes Design mit dem Strangler-Fig-Muster verwenden können, finden Sie unter Schrittweises Modernisieren älterer Microsoft ASP.NET (ASMX)-Webservices mithilfe von Containern und Amazon API Gateway.

DR

Siehe Disaster Recovery.

Erkennung von Driften

Verfolgung von Abweichungen von einer Basiskonfiguration Sie können es beispielsweise verwenden, AWS CloudFormation um Abweichungen bei den Systemressourcen zu erkennen, oder Sie können AWS Control Tower damit Änderungen in Ihrer landing zone erkennen, die sich auf die Einhaltung von Governance-Anforderungen auswirken könnten.

DVSM

Siehe Abbildung des Wertstroms in der Entwicklung.

E

EDA

Siehe explorative Datenanalyse.

Edge-Computing

Die Technologie, die die Rechenleistung für intelligente Geräte an den Rändern eines IoT-Netzwerks erhöht. Im Vergleich zu Cloud Computing kann Edge Computing die Kommunikationslatenz reduzieren und die Reaktionszeit verbessern.

Verschlüsselung

Ein Rechenprozess, der Klartextdaten, die für Menschen lesbar sind, in Chiffretext umwandelt.

Verschlüsselungsschlüssel

Eine kryptografische Zeichenfolge aus zufälligen Bits, die von einem Verschlüsselungsalgorithmus generiert wird. Schlüssel können unterschiedlich lang sein, und jeder Schlüssel ist so konzipiert, dass er unvorhersehbar und einzigartig ist.

Endianismus

Die Reihenfolge, in der Bytes im Computerspeicher gespeichert werden. Big-Endian-Systeme speichern das höchstwertige Byte zuerst. Little-Endian-Systeme speichern das niedrigwertigste Byte zuerst.

Endpunkt

Siehe Service-Endpunkt.

Endpunkt-Services

Ein Service, den Sie in einer Virtual Private Cloud (VPC) hosten können, um ihn mit anderen Benutzern zu teilen. Sie können einen Endpunktdienst mit anderen AWS-Konten oder AWS Identity and Access Management (IAM AWS PrivateLink -) Prinzipalen erstellen und diesen Berechtigungen gewähren. Diese Konten oder Prinzipale können sich privat mit Ihrem Endpunktservice verbinden, indem sie Schnittstellen-VPC-Endpunkte erstellen. Weitere Informationen finden Sie unter Einen Endpunkt-Service erstellen in der Amazon Virtual Private Cloud (Amazon VPC)-Dokumentation.

Unternehmensressourcenplanung (ERP)

Ein System, das wichtige Geschäftsprozesse (wie Buchhaltung, MES und Projektmanagement) für ein Unternehmen automatisiert und verwaltet.

Envelope-Verschlüsselung

Der Prozess der Verschlüsselung eines Verschlüsselungsschlüssels mit einem anderen Verschlüsselungsschlüssel. Weitere Informationen finden Sie unter Envelope-Verschlüsselung in der AWS Key Management Service (AWS KMS) -Dokumentation.

Umgebung

Eine Instance einer laufenden Anwendung. Die folgenden Arten von Umgebungen sind beim Cloud-Computing üblich:

- Entwicklungsumgebung – Eine Instance einer laufenden Anwendung, die nur dem Kernteam zur Verfügung steht, das für die Wartung der Anwendung verantwortlich ist. Entwicklungsumgebungen werden verwendet, um Änderungen zu testen, bevor sie in höhere Umgebungen übertragen werden. Diese Art von Umgebung wird manchmal als Testumgebung bezeichnet.
- Niedrigere Umgebungen – Alle Entwicklungsumgebungen für eine Anwendung, z. B. solche, die für erste Builds und Tests verwendet wurden.
- Produktionsumgebung – Eine Instance einer laufenden Anwendung, auf die Endbenutzer zugreifen können. In einer CI/CD-Pipeline ist die Produktionsumgebung die letzte Bereitstellungsumgebung.
- Höhere Umgebungen – Alle Umgebungen, auf die auch andere Benutzer als das Kernentwicklungsteam zugreifen können. Dies kann eine Produktionsumgebung, Vorproduktionsumgebungen und Umgebungen für Benutzerakzeptanztests umfassen.

Epics

In der agilen Methodik sind dies funktionale Kategorien, die Ihnen helfen, Ihre Arbeit zu organisieren und zu priorisieren. Epics bieten eine allgemeine Beschreibung der Anforderungen und Implementierungsaufgaben. Zu den Sicherheitsthemen AWS von CAF gehören beispielsweise Identitäts- und Zugriffsmanagement, Detektivkontrollen, Infrastruktursicherheit, Datenschutz und Reaktion auf Vorfälle. Weitere Informationen zu Epics in der AWS -Migrationsstrategie finden Sie im Leitfaden zur Programm-Implementierung.

ERP

Siehe Enterprise Resource Planning.

Explorative Datenanalyse (EDA)

Der Prozess der Analyse eines Datensatzes, um seine Hauptmerkmale zu verstehen. Sie sammeln oder aggregieren Daten und führen dann erste Untersuchungen durch, um Muster zu

finden, Anomalien zu erkennen und Annahmen zu überprüfen. EDA wird durchgeführt, indem zusammenfassende Statistiken berechnet und Datenvisualisierungen erstellt werden.

F

Faktentabelle

Die zentrale Tabelle in einem Sternschema. Sie speichert quantitative Daten über den Geschäftsbetrieb. In der Regel enthält eine Faktentabelle zwei Arten von Spalten: Spalten, die Kennzahlen enthalten, und Spalten, die einen Fremdschlüssel für eine Dimensionstabelle enthalten.

schnell scheitern

Eine Philosophie, die häufige und inkrementelle Tests verwendet, um den Entwicklungslebenszyklus zu verkürzen. Dies ist ein wichtiger Bestandteil eines agilen Ansatzes.

Grenze zur Fehlerisolierung

Dabei handelt es sich um eine Grenze AWS Cloud, z. B. eine Availability Zone AWS-Region, eine Steuerungsebene oder eine Datenebene, die die Auswirkungen eines Fehlers begrenzt und die Widerstandsfähigkeit von Workloads verbessert. Weitere Informationen finden Sie unter Grenzen zur AWS Fehlerisolierung.

Feature-Zweig

Siehe Zweig.

Features

Die Eingabedaten, die Sie verwenden, um eine Vorhersage zu treffen. In einem Fertigungskontext könnten Feature beispielsweise Bilder sein, die regelmäßig von der Fertigungslinie aus aufgenommen werden.

Bedeutung der Feature

Wie wichtig ein Feature für die Vorhersagen eines Modells ist. Dies wird in der Regel als numerischer Wert ausgedrückt, der mit verschiedenen Techniken wie Shapley Additive Explanations (SHAP) und integrierten Gradienten berechnet werden kann. Weitere Informationen finden Sie unter Interpretierbarkeit von Modellen für maschinelles Lernen mit:AWS.

Featuretransformation

Daten für den ML-Prozess optimieren, einschließlich der Anreicherung von Daten mit zusätzlichen Quellen, der Skalierung von Werten oder der Extraktion mehrerer Informationssätze aus einem einzigen Datenfeld. Das ermöglicht dem ML-Modell, von den Daten profitieren. Wenn Sie beispielsweise das Datum „27.05.2021 00:15:37“ in „2021“, „Mai“, „Donnerstag“ und „15“ aufschlüsseln, können Sie dem Lernalgorithmus helfen, nuancierte Muster zu erlernen, die mit verschiedenen Datenkomponenten verknüpft sind.

FGAC

Weitere Informationen finden Sie unter detaillierter Zugriffskontrolle.

Feinkörnige Zugriffskontrolle (FGAC)

Die Verwendung mehrerer Bedingungen, um eine Zugriffsanfrage zuzulassen oder abzulehnen.

Flash-Cut-Migration

Eine Datenbankmigrationsmethode, bei der eine kontinuierliche Datenreplikation durch Erfassung von Änderungsdaten verwendet wird, um Daten in kürzester Zeit zu migrieren, anstatt einen schrittweisen Ansatz zu verwenden. Ziel ist es, Ausfallzeiten auf ein Minimum zu beschränken.

G

Geoblocking

Siehe geografische Einschränkungen.

Geografische Einschränkungen (Geoblocking)

Bei Amazon eine Option CloudFront, um zu verhindern, dass Benutzer in bestimmten Ländern auf Inhaltsverteilungen zugreifen. Sie können eine Zulassungsliste oder eine Sperrliste verwenden, um zugelassene und gesperrte Länder anzugeben. Weitere Informationen finden Sie in der Dokumentation unter Beschränkung der geografischen Verteilung Ihrer Inhalte. CloudFront

Gitflow-Workflow

Ein Ansatz, bei dem niedrigere und höhere Umgebungen unterschiedliche Zweige in einem Quellcode-Repository verwenden. Der Gitflow-Workflow gilt als veraltet, und der Trunk-basierte Workflow ist der moderne, bevorzugte Ansatz.

Greenfield-Strategie

Das Fehlen vorhandener Infrastruktur in einer neuen Umgebung. Bei der Einführung einer Neuausrichtung einer Systemarchitektur können Sie alle neuen Technologien ohne Einschränkung der Kompatibilität mit der vorhandenen Infrastruktur auswählen, auch bekannt als Brownfield. Wenn Sie die bestehende Infrastruktur erweitern, könnten Sie Brownfield- und Greenfield-Strategien mischen.

Integritätsschutz

Eine allgemeine Regel, die dabei hilft, Ressourcen, Richtlinien und die Einhaltung von Vorschriften in allen Organisationseinheiten (OUs) zu regeln. Präventiver Integritätsschutz setzt Richtlinien durch, um die Einhaltung von Standards zu gewährleisten. Sie werden mithilfe von Service-Kontrollrichtlinien und IAM-Berechtigungsgrenzen implementiert. Detektivischer Integritätsschutz erkennt Richtlinienverstöße und Compliance-Probleme und generiert Warnmeldungen zur Abhilfe. Sie werden mithilfe von AWS Config, AWS Security Hub, Amazon GuardDuty AWS Trusted Advisor, Amazon Inspector und benutzerdefinierten AWS Lambda Prüfungen implementiert.

H

HEKTAR

Siehe Hochverfügbarkeit.

Heterogene Datenbankmigration

Migrieren Sie Ihre Quelldatenbank in eine Zieldatenbank, die eine andere Datenbank-Engine verwendet (z. B. Oracle zu Amazon Aurora). Eine heterogene Migration ist in der Regel Teil einer Neuarchitektur, und die Konvertierung des Schemas kann eine komplexe Aufgabe sein. AWS bietet AWS SCT, welches bei Schemakonvertierungen hilft.

hohe Verfügbarkeit (HA)

Die Fähigkeit eines Workloads, im Falle von Herausforderungen oder Katastrophen kontinuierlich und ohne Eingreifen zu arbeiten. HA-Systeme sind so konzipiert, dass sie automatisch ein Failover durchführen, gleichbleibend hohe Leistung bieten und unterschiedliche Lasten und Ausfälle mit minimalen Leistungseinbußen bewältigen.

historische Modernisierung

Ein Ansatz zur Modernisierung und Aufrüstung von Betriebstechnologiesystemen (OT), um den Bedürfnissen der Fertigungsindustrie besser gerecht zu werden. Ein Historian ist eine Art von Datenbank, die verwendet wird, um Daten aus verschiedenen Quellen in einer Fabrik zu sammeln und zu speichern.

Homogene Datenbankmigration

Migrieren Sie Ihre Quelldatenbank zu einer Zieldatenbank, die dieselbe Datenbank-Engine verwendet (z. B. Microsoft SQL Server zu Amazon RDS für SQL Server). Eine homogene Migration ist in der Regel Teil eines Hostwechsels oder eines Plattformwechsels. Sie können native Datenbankserviceprogramme verwenden, um das Schema zu migrieren.

heiße Daten

Daten, auf die häufig zugegriffen wird, z. B. Echtzeitdaten oder aktuelle Translationsdaten. Für diese Daten ist in der Regel eine leistungsstarke Speicherebene oder -klasse erforderlich, um schnelle Abfrageantworten zu ermöglichen.

Hotfix

Eine dringende Lösung für ein kritisches Problem in einer Produktionsumgebung. Aufgrund seiner Dringlichkeit wird ein Hotfix normalerweise außerhalb des typischen DevOps Release-Workflows erstellt.

Hypercare-Phase

Unmittelbar nach dem Cutover, der Zeitraum, in dem ein Migrationsteam die migrierten Anwendungen in der Cloud verwaltet und überwacht, um etwaige Probleme zu beheben. In der Regel dauert dieser Zeitraum 1–4 Tage. Am Ende der Hypercare-Phase überträgt das Migrationsteam in der Regel die Verantwortung für die Anwendungen an das Cloud-Betriebsteam.

I

IaC

Sehen Sie sich Infrastruktur als Code an.

Identitätsbasierte Richtlinie

Eine Richtlinie, die einem oder mehreren IAM-Prinzipalen zugeordnet ist und deren Berechtigungen innerhalb der AWS Cloud Umgebung definiert.

Leerlaufanwendung

Eine Anwendung mit einer durchschnittlichen CPU- und Arbeitsspeicherauslastung zwischen 5 und 20 Prozent über einen Zeitraum von 90 Tagen. In einem Migrationsprojekt ist es üblich, diese Anwendungen außer Betrieb zu nehmen oder sie On-Premises beizubehalten.

IIoT

Siehe Industrielles Internet der Dinge.

unveränderliche Infrastruktur

Ein Modell, das eine neue Infrastruktur für Produktionsworkloads bereitstellt, anstatt die bestehende Infrastruktur zu aktualisieren, zu patchen oder zu modifizieren. Unveränderliche Infrastrukturen sind von Natur aus konsistenter, zuverlässiger und vorhersehbarer als veränderliche Infrastrukturen. Weitere Informationen finden Sie in der Best Practice Deploy using immutable infrastructure im AWS Well-Architected Framework.

Eingehende (ingress) VPC

In einer Architektur AWS mit mehreren Konten ist dies eine VPC, die Netzwerkverbindungen von außerhalb einer Anwendung akzeptiert, überprüft und weiterleitet. Die AWS -Referenzarchitektur für die Sicherheit empfiehlt, Ihr Netzwerkkonto mit eingehenden und ausgehenden VPCs und Inspektions-VPCs einzurichten, um die bidirektionale Schnittstelle zwischen Ihrer Anwendung und dem Internet zu schützen.

Inkrementelle Migration

Eine Cutover-Strategie, bei der Sie Ihre Anwendung in kleinen Teilen migrieren, anstatt eine einziges vollständiges Cutover durchzuführen. Beispielsweise könnten Sie zunächst nur einige Microservices oder Benutzer auf das neue System umstellen. Nachdem Sie sich vergewissert haben, dass alles ordnungsgemäß funktioniert, können Sie weitere Microservices oder Benutzer schrittweise verschieben, bis Sie Ihr Legacy-System außer Betrieb nehmen können. Diese Strategie reduziert die mit großen Migrationen verbundenen Risiken.

Industrie 4.0

Ein Begriff, der 2016 von Klaus Schwab eingeführt wurde und sich auf die Modernisierung von Fertigungsprozessen durch Fortschritte in den Bereichen Konnektivität, Echtzeitdaten, Automatisierung, Analytik und KI/ML bezieht.

Infrastruktur

Alle Ressourcen und Komponenten, die in der Umgebung einer Anwendung enthalten sind.

Infrastructure as Code (IaC)

Der Prozess der Bereitstellung und Verwaltung der Infrastruktur einer Anwendung mithilfe einer Reihe von Konfigurationsdateien. IaC soll Ihnen helfen, das Infrastrukturmanagement zu zentralisieren, Ressourcen zu standardisieren und schnell zu skalieren, sodass neue Umgebungen wiederholbar, zuverlässig und konsistent sind.

Industrielles Internet der Dinge (IIoT)

Einsatz von mit dem Internet verbundenen Sensoren und Geräten in Industriesektoren wie Fertigung, Energie, Automobilindustrie, Gesundheitswesen, Biowissenschaften und Landwirtschaft. Mehr Informationen finden Sie unter Aufbau einer digitalen Transformationsstrategie für das industrielle Internet der Dinge (IIoT).

Inspektions-VPC

In einer Architektur AWS mit mehreren Konten eine zentralisierte VPC, die Inspektionen des Netzwerkverkehrs zwischen VPCs (in derselben oder unterschiedlichen AWS-Regionen), dem Internet und lokalen Netzwerken verwaltet. Die AWS -Referenzarchitektur für die Sicherheit empfiehlt, Ihr Netzwerkkonto mit eingehenden und ausgehenden VPCs und Inspektions-VPCs einzurichten, um die bidirektionale Schnittstelle zwischen Ihrer Anwendung und dem Internet zu schützen.

Internet of Things (IoT)

Das Netzwerk verbundener physischer Objekte mit eingebetteten Sensoren oder Prozessoren, das über das Internet oder über ein lokales Kommunikationsnetzwerk mit anderen Geräten und Systemen kommuniziert. Weitere Informationen finden Sie unter Was ist IoT?

Interpretierbarkeit

Ein Merkmal eines Modells für Machine Learning, das beschreibt, inwieweit ein Mensch verstehen kann, wie die Vorhersagen des Modells von seinen Eingaben abhängen. Weitere Informationen finden Sie unter Interpretierbarkeit von Modellen für Machine Learning mit AWS.

IoT

Siehe Internet der Dinge.

IT information library (ITIL, IT-Informationsbibliothek)

Eine Reihe von bewährten Methoden für die Bereitstellung von IT-Services und die Abstimmung dieser Services auf die Geschäftsanforderungen. ITIL bietet die Grundlage für ITSM.

T service management (ITSM, IT-Servicemanagement)

Aktivitäten im Zusammenhang mit der Gestaltung, Implementierung, Verwaltung und Unterstützung von IT-Services für eine Organisation. Informationen zur Integration von Cloud-Vorgängen mit ITSM-Tools finden Sie im Leitfaden zur Betriebsintegration.

BIS

Weitere Informationen finden Sie in der IT-Informationsbibliothek.

ITSM

Siehe IT-Servicemanagement.

L

Labelbasierte Zugangskontrolle (LBAC)

Eine Implementierung der Mandatory Access Control (MAC), bei der den Benutzern und den Daten selbst jeweils explizit ein Sicherheitslabelwert zugewiesen wird. Die Schnittmenge zwischen der Benutzersicherheitsbeschriftung und der Datensicherheitsbeschriftung bestimmt, welche Zeilen und Spalten für den Benutzer sichtbar sind.

Landing Zone

Eine landing zone ist eine gut strukturierte AWS Umgebung mit mehreren Konten, die skalierbar und sicher ist. Dies ist ein Ausgangspunkt, von dem aus Ihre Organisationen Workloads und Anwendungen schnell und mit Vertrauen in ihre Sicherheits- und Infrastrukturumgebung starten und bereitstellen können. Weitere Informationen zu Landing Zones finden Sie unter Einrichtung einer sicheren und skalierbaren AWS -Umgebung mit mehreren Konten..

Große Migration

Eine Migration von 300 oder mehr Servern.

SCHWARZ

Weitere Informationen finden Sie unter Label-basierte Zugriffskontrolle.

Geringste Berechtigung

Die bewährte Sicherheitsmethode, bei der nur die für die Durchführung einer Aufgabe erforderlichen Mindestberechtigungen erteilt werden. Weitere Informationen finden Sie unter Geringste Berechtigungen anwenden in der IAM-Dokumentation.

Lift and Shift

Siehe 7 Rs.

Little-Endian-System

Ein System, welches das niedrigwertigste Byte zuerst speichert. Siehe auch Endianness.

Niedrigere Umgebungen

Siehe Umwelt.

M

Machine Learning (ML)

Eine Art künstlicher Intelligenz, die Algorithmen und Techniken zur Mustererkennung und zum Lernen verwendet. ML analysiert aufgezeichnete Daten, wie z. B. Daten aus dem Internet der Dinge (IoT), und lernt daraus, um ein statistisches Modell auf der Grundlage von Mustern zu erstellen. Weitere Informationen finden Sie unter Machine Learning.

Hauptzweig

Siehe Filiale.

Malware

Software, die entwickelt wurde, um die Computersicherheit oder den Datenschutz zu gefährden. Malware kann Computersysteme stören, vertrauliche Informationen durchsickern lassen oder sich unbefugten Zugriff verschaffen. Beispiele für Malware sind Viren, Würmer, Ransomware, Trojaner, Spyware und Keylogger.

verwaltete Dienste

AWS -Services für die die Infrastrukturebene, das Betriebssystem und die Plattformen AWS betrieben werden, und Sie greifen auf die Endgeräte zu, um Daten zu speichern und abzurufen. Amazon Simple Storage Service (Amazon S3) und Amazon DynamoDB sind Beispiele für Managed Services. Diese werden auch als abstrakte Dienste bezeichnet.

Manufacturing Execution System (MES)

Ein Softwaresystem zur Nachverfolgung, Überwachung, Dokumentation und Steuerung von Produktionsprozessen, bei denen Rohstoffe in der Fertigung zu fertigen Produkten umgewandelt werden.

MAP

Siehe Migration Acceleration Program.

Mechanismus

Ein vollständiger Prozess, bei dem Sie ein Tool erstellen, die Akzeptanz des Tools vorantreiben und anschließend die Ergebnisse überprüfen, um Anpassungen vorzunehmen. Ein Mechanismus ist ein Zyklus, der sich im Laufe seiner Tätigkeit selbst verstärkt und verbessert. Weitere Informationen finden Sie unter Aufbau von Mechanismen im AWS Well-Architected Framework.

Mitgliedskonto

Alle AWS-Konten außer dem Verwaltungskonto, die Teil einer Organisation in sind. AWS Organizations Ein Konto kann jeweils nur einer Organisation angehören.

DURCHEINANDER

Siehe Manufacturing Execution System.

Message Queuing-Telemetrietransport (MQTT)

Ein leichtes machine-to-machine (M2M) -Kommunikationsprotokoll, das auf dem Publish/Subscribe-Muster für IoT-Geräte mit beschränkten Ressourcen basiert.

Microservice

Ein kleiner, unabhängiger Service, der über klar definierte APIs kommuniziert und in der Regel kleinen, eigenständigen Teams gehört. Ein Versicherungssystem kann beispielsweise Microservices beinhalten, die Geschäftsfunktionen wie Vertrieb oder Marketing oder Subdomains wie Einkauf, Schadenersatz oder Analytik zugeordnet sind. Zu den Vorteilen von Microservices gehören Agilität, flexible Skalierung, einfache Bereitstellung, wiederverwendbarer Code und Ausfallsicherheit. Weitere Informationen finden Sie unter Integration von Microservices mithilfe serverloser Dienste. AWS

Microservices-Architekturen

Ein Ansatz zur Erstellung einer Anwendung mit unabhängigen Komponenten, die jeden Anwendungsprozess als Microservice ausführen. Diese Microservices kommunizieren über eine klar definierte Schnittstelle mithilfe einfacher APIs. Jeder Microservice in dieser Architektur kann aktualisiert, bereitgestellt und skaliert werden, um den Bedarf an bestimmten Funktionen einer Anwendung zu decken. Weitere Informationen finden Sie unter Implementierung von Microservices auf. AWS

Migration Acceleration Program (MAP)

Ein AWS Programm, das Beratung, Unterstützung, Schulungen und Services bietet, um Unternehmen dabei zu unterstützen, eine solide betriebliche Grundlage für die Umstellung auf die Cloud zu schaffen und die anfänglichen Kosten von Migrationen auszugleichen. MAP umfasst eine Migrationsmethode für die methodische Durchführung von Legacy-Migrationen sowie eine Reihe von Tools zur Automatisierung und Beschleunigung gängiger Migrationsszenarien.

Migration in großem Maßstab

Der Prozess, bei dem der Großteil des Anwendungsportfolios in Wellen in die Cloud verlagert wird, wobei in jeder Welle mehr Anwendungen schneller migriert werden. In dieser Phase werden die bewährten Verfahren und Erkenntnisse aus den früheren Phasen zur Implementierung einer Migrationsfabrik von Teams, Tools und Prozessen zur Optimierung der Migration von Workloads durch Automatisierung und agile Bereitstellung verwendet. Dies ist die dritte Phase der AWS - Migrationsstrategie.

Migrationsfabrik

Funktionsübergreifende Teams, die die Migration von Workloads durch automatisierte, agile Ansätze optimieren. Zu den Teams in der Migrationsabteilung gehören in der Regel Betriebsabläufe, Geschäftsanalysten und Eigentümer, Migrationsingenieure, Entwickler und DevOps Experten, die in Sprints arbeiten. Zwischen 20 und 50 Prozent eines Unternehmensanwendungsportfolios bestehen aus sich wiederholenden Mustern, die durch einen Fabrik-Ansatz optimiert werden können. Weitere Informationen finden Sie in Diskussion über Migrationsfabriken und den Leitfaden zur Cloud-Migration-Fabrik in diesem Inhaltssatz.

Migrationsmetadaten

Die Informationen über die Anwendung und den Server, die für den Abschluss der Migration benötigt werden. Für jedes Migrationsmuster ist ein anderer Satz von Migrationsmetadaten erforderlich. Beispiele für Migrationsmetadaten sind das Zielsubnetz, die Sicherheitsgruppe und AWS das Konto.

Migrationsmuster

Eine wiederholbare Migrationsaufgabe, in der die Migrationsstrategie, das Migrationsziel und die verwendete Migrationsanwendung oder der verwendete Migrationsservice detailliert beschrieben werden. Beispiel: Rehost-Migration zu Amazon EC2 mit AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

Ein Online-Tool, das Informationen zur Validierung des Geschäftsszenarios für die Migration auf das bereitstellt. AWS Cloud MPA bietet eine detaillierte Portfoliobewertung (richtige Servergröße, Preisgestaltung, Gesamtbetriebskostenanalyse, Migrationskostenanalyse) sowie Migrationsplanung (Anwendungsdatenanalyse und Datenerfassung, Anwendungsgruppierung, Migrationspriorisierung und Wellenplanung). Das MPA-Tool (Anmeldung erforderlich) steht allen AWS Beratern und APN-Partnerberatern kostenlos zur Verfügung.

Migration Readiness Assessment (MRA)

Der Prozess, bei dem mithilfe des AWS CAF Erkenntnisse über den Cloud-Bereitschaftsstatus eines Unternehmens gewonnen, Stärken und Schwächen identifiziert und ein Aktionsplan zur Schließung festgestellter Lücken erstellt wird. Weitere Informationen finden Sie im Benutzerhandbuch für Migration Readiness. MRA ist die erste Phase der AWS - Migrationsstrategie.

Migrationsstrategie

Der Ansatz, der verwendet wurde, um einen Workload auf den AWS Cloud zu migrieren. Weitere Informationen finden Sie im Eintrag 7 Rs in diesem Glossar und unter Mobilisieren Sie Ihr Unternehmen, um umfangreiche Migrationen zu beschleunigen.

ML

Siehe maschinelles Lernen.

Modernisierung

Umwandlung einer veralteten (veralteten oder monolithischen) Anwendung und ihrer Infrastruktur in ein agiles, elastisches und hochverfügbares System in der Cloud, um Kosten zu senken, die Effizienz zu steigern und Innovationen zu nutzen. Weitere Informationen finden Sie unter Strategie zur Modernisierung von Anwendungen in der AWS Cloud.

Bewertung der Modernisierungsfähigkeit

Eine Bewertung, anhand derer festgestellt werden kann, ob die Anwendungen einer Organisation für die Modernisierung bereit sind, Vorteile, Risiken und Abhängigkeiten identifiziert und ermittelt wird, wie gut die Organisation den zukünftigen Status dieser Anwendungen unterstützen kann. Das Ergebnis der Bewertung ist eine Vorlage der Zielarchitektur, eine Roadmap, in der die Entwicklungsphasen und Meilensteine des Modernisierungsprozesses detailliert beschrieben werden, sowie ein Aktionsplan zur Behebung festgestellter Lücken. Weitere Informationen finden Sie unter Evaluierung der Modernisierungsbereitschaft von Anwendungen in der AWS Cloud.

Monolithische Anwendungen (Monolithen)

Anwendungen, die als ein einziger Service mit eng gekoppelten Prozessen ausgeführt werden. Monolithische Anwendungen haben verschiedene Nachteile. Wenn ein Anwendungs-Feature stark nachgefragt wird, muss die gesamte Architektur skaliert werden. Das Hinzufügen oder Verbessern der Feature einer monolithischen Anwendung wird ebenfalls komplexer, wenn die Codebasis wächst. Um diese Probleme zu beheben, können Sie eine Microservices-Architektur verwenden. Weitere Informationen finden Sie unter Zerlegen von Monolithen in Microservices.

MPA

Siehe Bewertung des Migrationsportfolios.

MQTT

Siehe Message Queuing-Telemetrietransport.

Mehrklassen-Klassifizierung

Ein Prozess, der dabei hilft, Vorhersagen für mehrere Klassen zu generieren (wobei eines von mehr als zwei Ergebnissen vorhergesagt wird). Ein ML-Modell könnte beispielsweise fragen: „Ist dieses Produkt ein Buch, ein Auto oder ein Telefon?“ oder „Welche Kategorie von Produkten ist für diesen Kunden am interessantesten?“

veränderbare Infrastruktur

Ein Modell, das die bestehende Infrastruktur für Produktionsworkloads aktualisiert und modifiziert. Für eine verbesserte Konsistenz, Zuverlässigkeit und Vorhersagbarkeit empfiehlt das AWS Well-Architected Framework die Verwendung einer unveränderlichen Infrastruktur als bewährte Methode.

O

OAC

Weitere Informationen finden Sie unter Origin Access Control.

OAI

Siehe Zugriffsidentität von Origin.

COM

Siehe organisatorisches Change-Management.

Offline-Migration

Eine Migrationsmethode, bei der der Quell-Workload während des Migrationsprozesses heruntergefahren wird. Diese Methode ist mit längeren Ausfallzeiten verbunden und wird in der Regel für kleine, unkritische Workloads verwendet.

OI

Siehe Betriebsintegration.

OLA

Siehe Vereinbarung auf operativer Ebene.

Online-Migration

Eine Migrationsmethode, bei der der Quell-Workload auf das Zielsystem kopiert wird, ohne offline genommen zu werden. Anwendungen, die mit dem Workload verbunden sind, können während der Migration weiterhin funktionieren. Diese Methode beinhaltet keine bis minimale Ausfallzeit und wird in der Regel für kritische Produktionsworkloads verwendet.

OPC-UA

Siehe Open Process Communications — Unified Architecture.

Offene Prozesskommunikation — Einheitliche Architektur (OPC-UA)

Ein machine-to-machine (M2M) -Kommunikationsprotokoll für die industrielle Automatisierung. OPC-UA bietet einen Interoperabilitätsstandard mit Datenverschlüsselungs-, Authentifizierungs- und Autorisierungsschemata.

Vereinbarung auf Betriebsebene (OLA)

Eine Vereinbarung, in der klargestellt wird, welche funktionalen IT-Gruppen sich gegenseitig versprechen zu liefern, um ein Service Level Agreement (SLA) zu unterstützen.

Überprüfung der Betriebsbereitschaft (ORR)

Eine Checkliste mit Fragen und zugehörigen bewährten Methoden, die Ihnen helfen, Vorfälle und mögliche Ausfälle zu verstehen, zu bewerten, zu verhindern oder deren Umfang zu reduzieren. Weitere Informationen finden Sie unter Operational Readiness Reviews (ORR) im AWS Well-Architected Framework.

Betriebstechnologie (OT)

Hardware- und Softwaresysteme, die mit der physischen Umgebung zusammenarbeiten, um industrielle Abläufe, Ausrüstung und Infrastruktur zu steuern. In der Fertigung ist die Integration

von OT- und Informationstechnologie (IT) -Systemen ein zentraler Schwerpunkt der Industrie 4.0-Transformationen.

Betriebsintegration (OI)

Der Prozess der Modernisierung von Abläufen in der Cloud, der Bereitschaftsplanung, Automatisierung und Integration umfasst. Weitere Informationen finden Sie im Leitfaden zur Betriebsintegration.

Organisationspfad

Ein Pfad, der von erstellt wird und in AWS CloudTrail dem alle Ereignisse für alle AWS-Konten in einer Organisation protokolliert werden. AWS Organizations Diese Spur wird in jedem AWS-Konto , der Teil der Organisation ist, erstellt und verfolgt die Aktivität in jedem Konto. Weitere Informationen finden Sie in der CloudTrail Dokumentation unter Erstellen eines Pfads für eine Organisation.

Organisatorisches Veränderungsmanagement (OCM)

Ein Framework für das Management wichtiger, disruptiver Geschäftstransformationen aus Sicht der Mitarbeiter, der Kultur und der Führung. OCM hilft Organisationen dabei, sich auf neue Systeme und Strategien vorzubereiten und auf diese umzustellen, indem es die Akzeptanz von Veränderungen beschleunigt, Übergangsprobleme angeht und kulturelle und organisatorische Veränderungen vorantreibt. In der AWS Migrationsstrategie wird dieses Framework aufgrund der Geschwindigkeit des Wandels, der bei Projekten zur Cloud-Einführung erforderlich ist, als Mitarbeiterbeschleunigung bezeichnet. Weitere Informationen finden Sie im OCM-Handbuch.

Ursprungszugriffskontrolle (OAC)

In CloudFront, eine erweiterte Option zur Zugriffsbeschränkung, um Ihre Amazon Simple Storage Service (Amazon S3) -Inhalte zu sichern. OAC unterstützt alle S3-Buckets insgesamt AWS-Regionen, serverseitige Verschlüsselung mit AWS KMS (SSE-KMS) sowie dynamische `PUT` und `DELETE` Anfragen an den S3-Bucket.

Ursprungszugriffsidentität (OAI)

In CloudFront, eine Option zur Zugriffsbeschränkung, um Ihre Amazon S3 S3-Inhalte zu sichern. Wenn Sie OAI verwenden, CloudFront erstellt es einen Principal, mit dem sich Amazon S3 authentifizieren kann. Authentifizierte Principals können nur über eine bestimmte Distribution auf Inhalte in einem S3-Bucket zugreifen. CloudFront Siehe auch OAC, das eine detailliertere und verbesserte Zugriffskontrolle bietet.

ODER

Siehe Überprüfung der Betriebsbereitschaft.

NICHT

Siehe Betriebstechnologie.

Ausgehende (egress) VPC

In einer Architektur AWS mit mehreren Konten eine VPC, die Netzwerkverbindungen verarbeitet, die von einer Anwendung aus initiiert werden. Die AWS -Referenzarchitektur für die Sicherheit empfiehlt, Ihr Netzwerkkonto mit eingehenden und ausgehenden VPCs und Inspektions-VPCs einzurichten, um die bidirektionale Schnittstelle zwischen Ihrer Anwendung und dem Internet zu schützen.

P

Berechtigungsgrenze

Eine IAM-Verwaltungsrichtlinie, die den IAM-Prinzipalen zugeordnet ist, um die maximalen Berechtigungen festzulegen, die der Benutzer oder die Rolle haben kann. Weitere Informationen finden Sie unter Berechtigungsgrenzen für IAM-Entitys in der IAM-Dokumentation.

persönlich identifizierbare Informationen (PII)

Informationen, die, wenn sie direkt betrachtet oder mit anderen verwandten Daten kombiniert werden, verwendet werden können, um vernünftige Rückschlüsse auf die Identität einer Person zu ziehen. Beispiele für personenbezogene Daten sind Namen, Adressen und Kontaktinformationen.

Personenbezogene Daten

Siehe persönlich identifizierbare Informationen.

Playbook

Eine Reihe vordefinierter Schritte, die die mit Migrationen verbundenen Aufgaben erfassen, z. B. die Bereitstellung zentraler Betriebsfunktionen in der Cloud. Ein Playbook kann die Form von Skripten, automatisierten Runbooks oder einer Zusammenfassung der Prozesse oder Schritte annehmen, die für den Betrieb Ihrer modernisierten Umgebung erforderlich sind.

PLC

Siehe programmierbare Logiksteuerung.

PLM

Siehe Produktlebenszyklusmanagement.

policy

Ein Objekt, das Berechtigungen definieren (siehe identitätsbasierte Richtlinie), Zugriffsbedingungen spezifizieren (siehe ressourcenbasierte Richtlinie) oder die maximalen Berechtigungen für alle Konten in einer Organisation definieren kann AWS Organizations (siehe Dienststeuerungsrichtlinie).

Polyglotte Beharrlichkeit

Unabhängige Auswahl der Datenspeichertechnologie eines Microservices auf der Grundlage von Datenzugriffsmustern und anderen Anforderungen. Wenn Ihre Microservices über dieselbe Datenspeichertechnologie verfügen, kann dies zu Implementierungsproblemen oder zu Leistungseinbußen führen. Microservices lassen sich leichter implementieren und erzielen eine bessere Leistung und Skalierbarkeit, wenn sie den Datenspeicher verwenden, der ihren Anforderungen am besten entspricht. Weitere Informationen finden Sie unter Datenpersistenz in Microservices aktivieren.

Portfoliobewertung

Ein Prozess, bei dem das Anwendungsportfolio ermittelt, analysiert und priorisiert wird, um die Migration zu planen. Weitere Informationen finden Sie in Bewerten der Migrationsbereitschaft.

predicate

Eine Abfragebedingung, die `true` oder zurückgibt`false`, was üblicherweise in einer Klausel vorkommt. `WHERE`

Prädikat Pushdown

Eine Technik zur Optimierung von Datenbankabfragen, bei der die Daten in der Abfrage vor der Übertragung gefiltert werden. Dadurch wird die Datenmenge reduziert, die aus der relationalen Datenbank abgerufen und verarbeitet werden muss, und die Abfrageleistung wird verbessert.

Präventive Kontrolle

Eine Sicherheitskontrolle, die verhindern soll, dass ein Ereignis eintritt. Diese Kontrollen stellen eine erste Verteidigungslinie dar, um unbefugten Zugriff oder unerwünschte Änderungen an Ihrem Netzwerk zu verhindern. Weitere Informationen finden Sie unter Präventive Kontrolle in Implementierung von Sicherheitskontrollen in AWS.

Prinzipal

Eine Entität AWS , die Aktionen ausführen und auf Ressourcen zugreifen kann. Bei dieser Entität handelt es sich in der Regel um einen Root-Benutzer für eine AWS-Konto, eine IAM-Rolle oder einen Benutzer. Weitere Informationen finden Sie unter Prinzipal in Rollenbegriffe und -konzepte in der IAM-Dokumentation.

Datenschutz durch Design

Ein Ansatz in der Systemtechnik, der den Datenschutz während des gesamten Engineering-Prozesses berücksichtigt.

Privat gehostete Zonen

Ein Container, der Informationen darüber enthält, wie Amazon Route 53 auf DNS-Abfragen für eine Domain und ihre Subdomains innerhalb einer oder mehrerer VPCs reagieren soll. Weitere Informationen finden Sie unter Arbeiten mit privat gehosteten Zonen in der Route-53-Dokumentation.

proaktive Steuerung

Eine Sicherheitskontrolle, die den Einsatz nicht richtlinienkonformer Ressourcen verhindern soll. Mit diesen Steuerelementen werden Ressourcen gescannt, bevor sie bereitgestellt werden. Wenn die Ressource nicht mit der Steuerung konform ist, wird sie nicht bereitgestellt. Weitere Informationen finden Sie im Referenzhandbuch zu Kontrollen in der AWS Control Tower Dokumentation und unter Proaktive Kontrollen unter Implementierung von Sicherheitskontrollen am AWS.

Produktlebenszyklusmanagement (PLM)

Das Management von Daten und Prozessen für ein Produkt während seines gesamten Lebenszyklus, vom Design, der Entwicklung und Markteinführung über Wachstum und Reife bis hin zur Markteinführung und Markteinführung.

Produktionsumgebung

Siehe Umgebung.

Speicherprogrammierbare Steuerung (SPS)

In der Fertigung ein äußerst zuverlässiger, anpassungsfähiger Computer, der Maschinen überwacht und Fertigungsprozesse automatisiert.

Pseudonymisierung

Der Prozess, bei dem persönliche Identifikatoren in einem Datensatz durch Platzhalterwerte ersetzt werden. Pseudonymisierung kann zum Schutz der Privatsphäre beitragen. Pseudonymisierte Daten gelten weiterhin als personenbezogene Daten.

veröffentlichen/abonnieren (pub/sub)

Ein Muster, das asynchrone Kommunikation zwischen Microservices ermöglicht, um die Skalierbarkeit und Reaktionsfähigkeit zu verbessern. In einem auf Microservices basierenden MES kann ein Microservice beispielsweise Ereignismeldungen in einem Kanal veröffentlichen, den andere Microservices abonnieren können. Das System kann neue Microservices hinzufügen, ohne den Veröffentlichungsservice zu ändern.

Q

Abfrageplan

Eine Reihe von Schritten, wie Anweisungen, die für den Zugriff auf die Daten in einem relationalen SQL-Datenbanksystem verwendet werden.

Abfrageplanregression

Wenn ein Datenbankserviceoptimierer einen weniger optimalen Plan wählt als vor einer bestimmten Änderung der Datenbankumgebung. Dies kann durch Änderungen an Statistiken, Beschränkungen, Umgebungseinstellungen, Abfrageparameter-Bindungen und Aktualisierungen der Datenbank-Engine verursacht werden.

R

RACI-Matrix

Siehe verantwortlich, rechenschaftspflichtig, konsultiert, informiert (RACI).

Ransomware

Eine bösartige Software, die entwickelt wurde, um den Zugriff auf ein Computersystem oder Daten zu blockieren, bis eine Zahlung erfolgt ist.

RASCI-Matrix

Siehe verantwortlich, rechenschaftspflichtig, konsultiert, informiert (RACI).

RCAC

Siehe Zugriffskontrolle für Zeilen und Spalten.

Read Replica

Eine Kopie einer Datenbank, die nur für Lesezwecke verwendet wird. Sie können Abfragen an das Lesereplikat weiterleiten, um die Belastung auf Ihrer Primärdatenbank zu reduzieren.

neu strukturieren

Siehe 7 Rs.

Recovery Point Objective (RPO)

Die maximal zulässige Zeitspanne seit dem letzten Datenwiederherstellungspunkt. Dies bestimmt, was als akzeptabler Datenverlust zwischen dem letzten Wiederherstellungspunkt und der Betriebsunterbrechung angesehen wird.

Wiederherstellungszeitziel (RTO)

Die maximal zulässige Verzögerung zwischen der Betriebsunterbrechung und der Wiederherstellung des Dienstes.

Refaktorierung

Siehe 7 Rs.

Region

Eine Sammlung von AWS Ressourcen in einem geografischen Gebiet. Jeder AWS-Region ist isoliert und unabhängig von den anderen, um Fehlertoleranz, Stabilität und Belastbarkeit zu gewährleisten. Weitere Informationen finden Sie unter Geben Sie an, was AWS-Regionen Ihr Konto verwenden kann.

Regression

Eine ML-Technik, die einen numerischen Wert vorhersagt. Zum Beispiel, um das Problem „Zu welchem Preis wird dieses Haus verkauft werden?“ zu lösen Ein ML-Modell könnte ein lineares Regressionsmodell verwenden, um den Verkaufspreis eines Hauses auf der Grundlage bekannter Fakten über das Haus (z. B. die Quadratmeterzahl) vorherzusagen.

rehosten

Siehe 7 Rs.

Veröffentlichung

In einem Bereitstellungsprozess der Akt der Förderung von Änderungen an einer Produktionsumgebung.

umziehen

Siehe 7 Rs.

neue Plattform

Siehe 7 Rs.

Rückkauf

Siehe 7 Rs.

Ausfallsicherheit

Die Fähigkeit einer Anwendung, Störungen zu widerstehen oder sich von ihnen zu erholen. Hochverfügbarkeit und Notfallwiederherstellung sind häufig Überlegungen bei der Planung der Ausfallsicherheit in der. AWS Cloud Weitere Informationen finden Sie unter AWS Cloud Resilienz.

Ressourcenbasierte Richtlinie

Eine mit einer Ressource verknüpfte Richtlinie, z. B. ein Amazon-S3-Bucket, ein Endpunkt oder ein Verschlüsselungsschlüssel. Diese Art von Richtlinie legt fest, welchen Prinzipalen der Zugriff gewährt wird, welche Aktionen unterstützt werden und welche anderen Bedingungen erfüllt sein müssen.

RACI-Matrix (verantwortlich, rechenschaftspflichtig, konsultiert, informiert)

Eine Matrix, die die Rollen und Verantwortlichkeiten aller an Migrationsaktivitäten und Cloud-Operationen beteiligten Parteien definiert. Der Matrixname leitet sich von den in der Matrix definierten Zuständigkeitstypen ab: verantwortlich (R), rechenschaftspflichtig (A), konsultiert (C) und informiert (I). Der Unterstützungstyp (S) ist optional. Wenn Sie Unterstützung einbeziehen, wird die Matrix als RASCI-Matrix bezeichnet, und wenn Sie sie ausschließen, wird sie als RACI-Matrix bezeichnet.

Reaktive Kontrolle

Eine Sicherheitskontrolle, die darauf ausgelegt ist, die Behebung unerwünschter Ereignisse oder Abweichungen von Ihren Sicherheitsstandards voranzutreiben. Weitere Informationen finden Sie unter Reaktive Kontrolle in Implementieren von Sicherheitskontrollen in AWS.

Beibehaltung

Siehe 7 Rs.

zurückziehen

Siehe 7 Rs.

Drehung

Der Vorgang, bei dem ein Geheimnis regelmäßig aktualisiert wird, um es einem Angreifer zu erschweren, auf die Anmeldeinformationen zuzugreifen.

Zugriffskontrolle für Zeilen und Spalten (RCAC)

Die Verwendung einfacher, flexibler SQL-Ausdrücke mit definierten Zugriffsregeln. RCAC besteht aus Zeilenberechtigungen und Spaltenmasken.

RPO

Siehe Recovery Point Objective.

RTO

Siehe Ziel der Wiederherstellungszeit.

Runbook

Eine Reihe manueller oder automatisierter Verfahren, die zur Ausführung einer bestimmten Aufgabe erforderlich sind. Diese sind in der Regel darauf ausgelegt, sich wiederholende Operationen oder Verfahren mit hohen Fehlerquoten zu rationalisieren.

S

SAML 2.0

Ein offener Standard, den viele Identitätsanbieter (IdPs) verwenden. Diese Funktion ermöglicht föderiertes Single Sign-On (SSO), sodass sich Benutzer bei den API-Vorgängen anmelden AWS Management Console oder die AWS API-Operationen aufrufen können, ohne dass Sie einen Benutzer in IAM für alle in Ihrer Organisation erstellen müssen. Weitere Informationen zum SAML-2.0.-basierten Verbund finden Sie unter Über den SAML-2.0-basierten Verbund in der IAM-Dokumentation.

SCADA

Siehe Aufsichtskontrolle und Datenerfassung.

SCP

Siehe Richtlinie zur Dienstkontrolle.

Secret

Interne AWS Secrets Manager, vertrauliche oder eingeschränkte Informationen, wie z. B. ein Passwort oder Benutzeranmeldeinformationen, die Sie in verschlüsselter Form speichern. Es besteht aus dem geheimen Wert und seinen Metadaten. Der geheime Wert kann binär, eine einzelne Zeichenfolge oder mehrere Zeichenketten sein. Weitere Informationen finden Sie unter Was ist in einem Secrets Manager Manager-Geheimnis? in der Secrets Manager Manager-Dokumentation.

Sicherheitskontrolle

Ein technischer oder administrativer Integritätsschutz, der die Fähigkeit eines Bedrohungsakteurs, eine Schwachstelle auszunutzen, verhindert, erkennt oder einschränkt. Es gibt vier Haupttypen von Sicherheitskontrollen: präventiv, detektiv, reaktionsschnell und proaktiv.

Härtung der Sicherheit

Der Prozess, bei dem die Angriffsfläche reduziert wird, um sie widerstandsfähiger gegen Angriffe zu machen. Dies kann Aktionen wie das Entfernen von Ressourcen, die nicht mehr benötigt werden, die Implementierung der bewährten Sicherheitsmethode der Gewährung geringster Berechtigungen oder die Deaktivierung unnötiger Feature in Konfigurationsdateien umfassen.

System zur Verwaltung von Sicherheitsinformationen und Ereignissen (security information and event management – SIEM)

Tools und Services, die Systeme für das Sicherheitsinformationsmanagement (SIM) und das Management von Sicherheitsereignissen (SEM) kombinieren. Ein SIEM-System sammelt, überwacht und analysiert Daten von Servern, Netzwerken, Geräten und anderen Quellen, um Bedrohungen und Sicherheitsverletzungen zu erkennen und Warnmeldungen zu generieren.

Automatisierung von Sicherheitsreaktionen

Eine vordefinierte und programmierte Aktion, die darauf ausgelegt ist, automatisch auf ein Sicherheitsereignis zu reagieren oder es zu beheben. Diese Automatisierungen dienen als detektive oder reaktionsschnelle Sicherheitskontrollen, die Sie bei der Implementierung bewährter AWS Sicherheitsmethoden unterstützen. Beispiele für automatisierte Antwortaktionen sind das Ändern einer VPC-Sicherheitsgruppe, das Patchen einer Amazon EC2 EC2-Instance oder das Rotieren von Anmeldeinformationen.

Serverseitige Verschlüsselung

Verschlüsselung von Daten am Zielort durch denjenigen AWS -Service , der sie empfängt.

Service-Kontrollrichtlinie (SCP)

Eine Richtlinie, die eine zentrale Kontrolle über die Berechtigungen für alle Konten in einer Organisation in AWS Organizations ermöglicht. SCPs definieren Integritätsschutz oder legen Grenzwerte für Aktionen fest, die ein Administrator an Benutzer oder Rollen delegieren kann. Sie können SCPs als Zulassungs- oder Ablehnungslisten verwenden, um festzulegen, welche Services oder Aktionen zulässig oder verboten sind. Weitere Informationen finden Sie in der AWS Organizations Dokumentation unter Richtlinien zur Dienststeuerung.

Service-Endpunkt

Die URL des Einstiegspunkts für einen AWS -Service. Sie können den Endpunkt verwenden, um programmgesteuert eine Verbindung zum Zielservice herzustellen. Weitere Informationen finden Sie unter AWS -Service -Endpunkte in der Allgemeine AWS-Referenz.

Service Level Agreement (SLA)

Eine Vereinbarung, in der klargestellt wird, was ein IT-Team seinen Kunden zu bieten verspricht, z. B. in Bezug auf Verfügbarkeit und Leistung der Services.

Service-Level-Indikator (SLI)

Eine Messung eines Leistungsaspekts eines Dienstes, z. B. seiner Fehlerrate, Verfügbarkeit oder Durchsatz.

Service-Level-Ziel (SLO)

Eine Zielkennzahl, die den Zustand eines Dienstes darstellt, gemessen anhand eines Service-Level-Indikators.

Modell der geteilten Verantwortung

Ein Modell, das die Verantwortung beschreibt, mit der Sie gemeinsam AWS für Cloud-Sicherheit und Compliance verantwortlich sind. AWS ist für die Sicherheit der Cloud verantwortlich, wohingegen Sie für die Sicherheit in der Cloud verantwortlich sind. Weitere Informationen finden Sie unter Modell der geteilten Verantwortung.

SIEM

Siehe Sicherheitsinformations- und Event-Management-System.

Single Point of Failure (SPOF)

Ein Fehler in einer einzelnen, kritischen Komponente einer Anwendung, der das System stören kann.

SLA

Siehe Service Level Agreement.

SLI

Siehe Service-Level-Indikator.

ALSO

Siehe Service-Level-Ziel.

split-and-seed Modell

Ein Muster für die Skalierung und Beschleunigung von Modernisierungsprojekten. Sobald neue Features und Produktversionen definiert werden, teilt sich das Kernteam auf, um neue Produktteams zu bilden. Dies trägt zur Skalierung der Fähigkeiten und Services Ihrer Organisation bei, verbessert die Produktivität der Entwickler und unterstützt schnelle Innovationen. Weitere Informationen finden Sie unter Schrittweiser Ansatz zur Modernisierung von Anwendungen in der AWS Cloud.

SPOTTEN

Siehe Single Point of Failure.

Sternschema

Eine Datenbank-Organisationsstruktur, die eine große Faktentabelle zum Speichern von Transaktions- oder Messdaten und eine oder mehrere kleinere dimensionale Tabellen zum Speichern von Datenattributen verwendet. Diese Struktur ist für die Verwendung in einem Data Warehouse oder für Business Intelligence-Zwecke konzipiert.

Strangler-Fig-Muster

Ein Ansatz zur Modernisierung monolithischer Systeme, bei dem die Systemfunktionen schrittweise umgeschrieben und ersetzt werden, bis das Legacy-System außer Betrieb genommen werden kann. Dieses Muster verwendet die Analogie einer Feigenrebe, die zu einem etablierten Baum heranwächst und schließlich ihren Wirt überwindet und ersetzt. Das Muster wurde eingeführt von Martin Fowler als Möglichkeit, Risiken beim Umschreiben monolithischer Systeme zu managen. Ein Beispiel für die Anwendung dieses Musters finden Sie

unter Schrittweises Modernisieren älterer Microsoft ASP.NET (ASMX)-Webservices mithilfe von Containern und Amazon API Gateway.

Subnetz

Ein Bereich von IP-Adressen in Ihrer VPC. Ein Subnetz muss sich in einer einzigen Availability Zone befinden.

Aufsichtskontrolle und Datenerfassung (SCADA)

In der Fertigung ein System, das Hardware und Software zur Überwachung von Sachanlagen und Produktionsabläufen verwendet.

Symmetrische Verschlüsselung

Ein Verschlüsselungsalgorithmus, der denselben Schlüssel zum Verschlüsseln und Entschlüsseln der Daten verwendet.

synthetisches Testen

Testen eines Systems auf eine Weise, die Benutzerinteraktionen simuliert, um potenzielle Probleme zu erkennen oder die Leistung zu überwachen. Sie können Amazon CloudWatch Synthetics verwenden, um diese Tests zu erstellen.

T

tags

Schlüssel-Wert-Paare, die als Metadaten für die Organisation Ihrer Ressourcen dienen. AWS Mit Tags können Sie Ressourcen verwalten, identifizieren, organisieren, suchen und filtern. Weitere Informationen finden Sie unter Markieren Ihrer AWS -Ressourcen.

Zielvariable

Der Wert, den Sie in überwachtem ML vorhersagen möchten. Dies wird auch als Ergebnisvariable bezeichnet. In einer Fertigungsumgebung könnte die Zielvariable beispielsweise ein Produktfehler sein.

Aufgabenliste

Ein Tool, das verwendet wird, um den Fortschritt anhand eines Runbooks zu verfolgen. Eine Aufgabenliste enthält eine Übersicht über das Runbook und eine Liste mit allgemeinen Aufgaben, die erledigt werden müssen. Für jede allgemeine Aufgabe werden der geschätzte Zeitaufwand, der Eigentümer und der Fortschritt angegeben.

Testumgebungen

Siehe Umgebung.

Training

Daten für Ihr ML-Modell bereitstellen, aus denen es lernen kann. Die Trainingsdaten müssen die richtige Antwort enthalten. Der Lernalgorithmus findet Muster in den Trainingsdaten, die die Attribute der Input-Daten dem Ziel (die Antwort, die Sie voraussagen möchten) zuordnen. Es gibt ein ML-Modell aus, das diese Muster erfasst. Sie können dann das ML-Modell verwenden, um Voraussagen für neue Daten zu erhalten, bei denen Sie das Ziel nicht kennen.

Transit-Gateway

Ein Transit-Gateway ist ein Netzwerk-Transit-Hub, mit dem Sie Ihre VPCs und On-Premises-Netzwerke miteinander verbinden können. Weitere Informationen finden Sie in der AWS Transit Gateway Dokumentation unter Was ist ein Transit-Gateway.

Stammbasierter Workflow

Ein Ansatz, bei dem Entwickler Feature lokal in einem Feature-Zweig erstellen und testen und diese Änderungen dann im Hauptzweig zusammenführen. Der Hauptzweig wird dann sequentiell für die Entwicklungs-, Vorproduktions- und Produktionsumgebungen erstellt.

Vertrauenswürdiger Zugriff

Gewährung von Berechtigungen für einen Dienst, den Sie angeben, um Aufgaben in Ihrer Organisation AWS Organizations und in deren Konten in Ihrem Namen auszuführen. Der vertrauenswürdige Service erstellt in jedem Konto eine mit dem Service verknüpfte Rolle, wenn diese Rolle benötigt wird, um Verwaltungsaufgaben für Sie auszuführen. Weitere Informationen finden Sie in der AWS Organizations Dokumentation unter Verwendung AWS Organizations mit anderen AWS Diensten.

Optimieren

Aspekte Ihres Trainingsprozesses ändern, um die Genauigkeit des ML-Modells zu verbessern. Sie können das ML-Modell z. B. trainieren, indem Sie einen Beschriftungssatz generieren, Beschriftungen hinzufügen und diese Schritte dann mehrmals unter verschiedenen Einstellungen wiederholen, um das Modell zu optimieren.

Zwei-Pizzen-Team

Ein kleines DevOps Team, das Sie mit zwei Pizzen ernähren können. Eine Teamgröße von zwei Pizzen gewährleistet die bestmögliche Gelegenheit zur Zusammenarbeit bei der Softwareentwicklung.

U

Unsicherheit

Ein Konzept, das sich auf ungenaue, unvollständige oder unbekannte Informationen bezieht, die die Zuverlässigkeit von prädiktiven ML-Modellen untergraben können. Es gibt zwei Arten von Unsicherheit: Epistemische Unsicherheit wird durch begrenzte, unvollständige Daten verursacht, wohingegen aleatorische Unsicherheit durch Rauschen und Randomisierung verursacht wird, die in den Daten liegt. Weitere Informationen finden Sie im Leitfaden Quantifizieren der Unsicherheit in Deep-Learning-Systemen.

undifferenzierte Aufgaben

Diese Arbeit wird auch als Schwerstarbeit bezeichnet. Dabei handelt es sich um Arbeiten, die zwar für die Erstellung und den Betrieb einer Anwendung erforderlich sind, aber dem Endbenutzer keinen direkten Mehrwert bieten oder keinen Wettbewerbsvorteil bieten. Beispiele für undifferenzierte Aufgaben sind Beschaffung, Wartung und Kapazitätsplanung.

höhere Umgebungen

Siehe Umgebung.

V

Vacuuming

Ein Vorgang zur Datenbankwartung, bei dem die Datenbank nach inkrementellen Aktualisierungen bereinigt wird, um Speicherplatz zurückzugewinnen und die Leistung zu verbessern.

Versionskontrolle

Prozesse und Tools zur Nachverfolgung von Änderungen, z. B. Änderungen am Quellcode in einem Repository.

VPC-Peering

Eine Verbindung zwischen zwei VPCs, mit der Sie den Datenverkehr mithilfe von privaten IP-Adressen weiterleiten können. Weitere Informationen finden Sie unter Was ist VPC-Peering? in der Amazon-VPC-Dokumentation.

Schwachstelle

Ein Software- oder Hardwarefehler, der die Sicherheit des Systems gefährdet.

W

Warmer Cache

Ein Puffer-Cache, der aktuelle, relevante Daten enthält, auf die häufig zugegriffen wird. Die Datenbank-Instance kann aus dem Puffer-Cache lesen, was schneller ist als das Lesen aus dem Hauptspeicher oder von der Festplatte.

warme Daten

Daten, auf die selten zugegriffen wird. Bei der Abfrage dieser Art von Daten sind mäßig langsame Abfragen in der Regel akzeptabel.

Fensterfunktion

Eine SQL-Funktion, die eine Berechnung für eine Gruppe von Zeilen durchführt, die sich in irgendeiner Weise auf den aktuellen Datensatz beziehen. Fensterfunktionen sind nützlich für die Verarbeitung von Aufgaben wie die Berechnung eines gleitenden Durchschnitts oder für den Zugriff auf den Wert von Zeilen auf der Grundlage der relativen Position der aktuellen Zeile.

Workload

Ein Workload ist eine Sammlung von Ressourcen und Code, die einen Unternehmenswert bietet, wie z. B. eine kundenorientierte Anwendung oder ein Backend-Prozess.

Workstream

Funktionsgruppen in einem Migrationsprojekt, die für eine bestimmte Reihe von Aufgaben verantwortlich sind. Jeder Workstream ist unabhängig, unterstützt aber die anderen Workstreams im Projekt. Der Portfolio-Workstream ist beispielsweise für die Priorisierung von Anwendungen, die Wellenplanung und die Erfassung von Migrationsmetadaten verantwortlich. Der Portfolio-Workstream liefert diese Komponenten an den Migrations-Workstream, der dann die Server und Anwendungen migriert.

WURM

Sehen Sie einmal schreiben, viele lesen.

WQF

Weitere Informationen finden Sie unter AWS Workload Qualification Framework.

einmal schreiben, viele lesen (WORM)

Ein Speichermodell, das Daten ein einziges Mal schreibt und verhindert, dass die Daten gelöscht oder geändert werden. Autorisierte Benutzer können die Daten so oft wie nötig lesen, aber sie können sie nicht ändern. Diese Datenspeicherinfrastruktur gilt als unveränderlich.

Z

Zero-Day-Exploit

Ein Angriff, in der Regel Malware, der eine Zero-Day-Sicherheitslücke ausnutzt.

Zero-Day-Sicherheitslücke

Ein unfehlbarer Fehler oder eine Sicherheitslücke in einem Produktionssystem. Bedrohungsakteure können diese Art von Sicherheitslücke nutzen, um das System anzugreifen. Entwickler werden aufgrund des Angriffs häufig auf die Sicherheitsanfälligkeit aufmerksam.

Zombie-Anwendung

Eine Anwendung, deren durchschnittliche CPU- und Arbeitsspeichernutzung unter 5 Prozent liegt. In einem Migrationsprojekt ist es üblich, diese Anwendungen außer Betrieb zu nehmen.

Die vorliegende Übersetzung wurde maschinell erstellt. Im Falle eines Konflikts oder eines Widerspruchs zwischen dieser übersetzten Fassung und der englischen Fassung (einschließlich infolge von Verzögerungen bei der Übersetzung) ist die englische Fassung maßgeblich.