



클라우드 설계 패턴, 아키텍처 및 구현

AWS 규범적 지침



AWS 규범적 지침: 클라우드 설계 패턴, 아키텍처 및 구현

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon의 상표 및 브랜드 디자인은 Amazon 외 제품 또는 서비스와 함께, 브랜드 이미지를 떨어뜨리거나 고객에게 혼동을 일으킬 수 있는 방식으로 사용할 수 없습니다. Amazon이 소유하지 않은 기타 모든 상표는 Amazon과 제휴 관계이거나 관련이 있거나 후원 관계와 관계없이 해당 소유자의 자산입니다.

Table of Contents

소개	1
목표 비즈니스 성과	2
손상 방지 레이어 패턴	3
Intent	3
동기부여	3
적용 가능성	3
문제 및 고려 사항	3
구현	4
고수준 아키텍처	4
를 사용한 구현AWS서비스	6
샘플 코드	7
GitHub저장소	8
관련 콘텐츠	8
API 라우팅 패턴	9
호스트 이름 라우팅	9
일반적인 사용 사례	9
장점	10
단점	10
경로 라우팅	10
일반적인 사용 사례	11
HTTP 서비스 역방향 프록시	11
API Gateway	13
CloudFront	14
HTTP 헤더 라우팅	15
장점	16
단점	17
회로 차단기 패턴	18
의도	18
목적	18
적용 가능성	19
문제 및 고려 사항	19
구현	20
전반적인 아키텍처	20
서비스를 이용한 AWS 구현	21

샘플 코드	22
GitHub 리포지토리	23
블로그 참조	23
관련 콘텐츠	24
이벤트 소싱 패턴	25
Intent	25
목적	25
적용 가능성	25
문제 및 고려 사항	25
구현	27
전반적인 아키텍처	27
AWS 서비스를 사용한 구현	29
블로그 참조	31
육각형 아키텍처 패턴	32
의도	32
목적	32
적용 가능성	32
문제 및 고려 사항	33
구현	33
상위 수준 아키텍처	34
를 사용한 구현 AWS 서비스	34
샘플 코드	35
관련 콘텐츠	39
비디오	39
게시-구독 패턴	40
Intent	40
목적	40
적용 가능성	40
문제 및 고려 사항	40
구현	42
전반적인 아키텍처	42
AWS 서비스를 사용한 구현	42
워크숍	45
블로그 참조	45
관련 콘텐츠	45
백오프 패턴으로 재시도	46

Intent	46
동기부여	46
적용 가능성	46
문제 및 고려 사항	46
구현	47
고수준 아키텍처	47
를 사용한 구현AWS서비스	47
샘플 코드	48
GitHub저장소	49
관련 콘텐츠	49
Saga 패턴	50
Saga 코레오그래피	51
Saga 오케스트레이션	51
Saga 코레오그래피	52
의도	52
목적	52
적용 가능성	53
문제 및 고려 사항	53
구현	54
관련 콘텐츠	57
Saga 오케스트레이션	57
의도	57
목적	57
적용 가능성	58
문제 및 고려 사항	58
구현	59
블로그 참조	63
관련 콘텐츠	64
비디오	64
스캐터 게더링 패턴	65
의도	65
목적	65
적용 가능성	65
문제 및 고려 사항	66
구현	66
전반적인 아키텍처	66

구현은 다음을 사용합니다. AWS 서비스	69
워크숍	72
블로그 참조	72
관련 콘텐츠	72
스트랭글러 무화과 패턴	73
Intent	73
동기 부여	73
적용 가능성	74
문제 및 고려 사항	74
구현	75
고수준 아키텍처	75
AWS서비스를 사용한 구현	80
워크숍	84
블로그 레퍼런스	84
관련 콘텐츠	84
트랜잭션 아웃박스 패턴	85
의도	85
목적	85
적용 가능성	85
문제 및 고려 사항	85
구현	86
전반적인 아키텍처	86
AWS 서비스를 사용한 구현	86
샘플 코드	91
아웃박스 테이블 사용	91
변경 데이터 캡처 (CDC) 사용	92
GitHub 리포지토리	94
리소스	95
문서 기록	96
용어집	98
#	98
A	99
B	101
C	103
D	106
E	110

F	112
G	113
H	114
I	115
L	117
M	118
O	122
P	124
Q	126
R	127
S	129
T	133
U	134
V	134
W	135
Z	136
.....	CXXXVII

클라우드 설계 패턴, 아키텍처 및 구현

Anitha Deenadayalan, Amazon Web Services(AWS)

2024년 5월 ([문서 기록](#))

이 가이드는 서비스를 사용하여 AWS 일반적으로 사용되는 현대화 설계 패턴을 구현하기 위한 지침을 제공합니다. 확장성을 달성하고, 릴리스 속도를 개선하고, 변경 사항의 영향 범위를 줄이고, 회귀를 줄이기 위해 마이크로서비스 아키텍처를 활용하여 설계되는 최신 애플리케이션이 갈수록 늘어나고 있습니다. 덕분에 개발자 생산성이 향상되고 민첩성이 높아지며 혁신이 개선되고 비즈니스 요구 사항에 대한 집중도가 향상되고 있습니다. 또한 마이크로서비스 아키텍처는 서비스와 데이터베이스를 위한 최상의 기술 활용 방식을 지원하고, 다국어 코드 및 다중 언어 지속성을 증진합니다.

일반적으로 모놀리식 애플리케이션은 단일 프로세스에서 실행되고, 단일 데이터 스토어를 사용하며, 수직적으로 규모 조정되는 서버에서 실행됩니다. 이에 비해 현대적 마이크로서비스 애플리케이션은 세분화되고 독립적인 장애 도메인을 가지고 있고, 네트워크에서 서비스로 실행되며, 사용 사례에 따라 둘 이상의 데이터 스토어를 사용할 수 있습니다. 서비스는 수평적으로 규모 조정되며 단일 트랜잭션이 여러 데이터베이스에 걸쳐 실행될 수 있습니다. 개발 팀은 마이크로서비스 아키텍처를 사용하여 애플리케이션을 개발할 때 데이터 스토어 전반의 네트워크 통신, 다국어 지속성, 수평적 규모 조정, 최종 일관성 및 트랜잭션 처리에 중점을 두어야 합니다. 따라서 모더니제이션 패턴은 최신 애플리케이션 개발에서 흔히 발생하는 문제를 해결하는 데 중요하며 소프트웨어 제공을 가속화하는 데 도움이 됩니다.

이 가이드에서는 Well-Architected 모범 사례를 바탕으로 설계 패턴에 적합한 클라우드 아키텍처를 선택하려는 클라우드 아키텍트, 기술 책임자, 애플리케이션 및 비즈니스 소유자, 개발자를 위한 기술 참조를 제공합니다. 이 가이드에서 설명하는 각 패턴은 마이크로서비스 아키텍처에서 알려진 하나 이상의 시나리오를 다룹니다. 이 가이드에서는 각 패턴과 관련된 문제 및 고려 사항을 설명하고, 전반적인 아키텍처 구현을 보여주며, 패턴의 AWS 구현을 설명합니다. 가능한 경우 오픈 소스 GitHub 샘플 및 위크숍 링크가 제공됩니다.

이 가이드에서는 다음과 같은 패턴을 다룹니다.

- [손상 방지 계층](#)
- [API 라우팅 패턴](#):
 - [호스트 이름 라우팅](#)
 - [경로 라우팅](#)
 - [HTTP 헤더 라우팅](#)
- [회로 차단기](#)

- [이벤트 소싱](#)
- [육각형 아키텍처](#)
- [구독-게시](#)
- [백오프를 사용한 재시도](#)
- [Saga 패턴:](#)
 - [Saga 코레오그래피](#)
 - [Saga 오케스트레이션](#)
- [스캐터 게더링](#)
- [스트랭글러 피그](#)
- [트랜잭션 아웃박스](#)

목표 비즈니스 성과

이 가이드에서 설명하는 패턴을 사용하여 애플리케이션을 현대화하면 다음과 같은 이점을 얻을 수 있습니다.

- 비용 및 성능에 최적화된 신뢰성이 높고 안전하며 운영 효율적인 아키텍처를 설계하고 구현합니다.
- 이러한 패턴이 필요한 사용 사례의 주기 시간을 줄이면, 그만큼 조직의 중요한 과제에 집중할 수 있습니다.
- AWS 서비스를 사용하여 패턴 구현을 표준화함으로써 개발을 가속화합니다.
- 개발자가 기술적 부채를 물려받지 않고 최신 애플리케이션을 구축할 수 있도록 지원합니다.

손상 방지 레이어 패턴

Intent

손상 방지 계층 (ACL) 패턴은 도메인 모델 시맨틱스를 한 시스템에서 다른 시스템으로 변환하는 중재 계층 역할을 합니다. 업스트림 팀이 설정한 통신 계약을 사용하기 전에 업스트림 경계 컨텍스트 (모노리스) 모델을 다운스트림 경계 컨텍스트 (마이크로서비스) 에 적합한 모델로 변환합니다. 이 패턴은 다운스트림 바운드 컨텍스트에 핵심 하위 도메인이 포함되어 있거나 업스트림 모델이 수정할 수 없는 레거시 시스템인 경우에 적용할 수 있습니다. 또한 통화를 대상 시스템으로 투명하게 리디렉션해야 할 때 발신자가 변경되지 않도록 하여 변환 위험과 비즈니스 중단을 줄여줍니다.

동기부여

마이그레이션 프로세스 중에 모놀리식 애플리케이션을 마이크로서비스로 마이그레이션할 때 새로 마이그레이션된 서비스의 도메인 모델 시맨틱이 변경될 수 있습니다. 이러한 마이크로서비스를 호출하기 위해 모놀리스 내의 기능이 필요한 경우 호출 서비스를 변경할 필요 없이 통화를 마이그레이션된 서비스로 라우팅해야 합니다. ACL 패턴을 사용하면 모놀리스가 호출을 새로운 시맨틱으로 변환하는 어댑터 또는 파사드 레이어 역할을 하여 마이크로서비스를 투명하게 호출할 수 있습니다.

적용 가능성

다음과 같은 경우 이 패턴을 사용하는 것이 좋습니다.

- 기존 모놀리식 애플리케이션은 마이크로서비스로 마이그레이션된 함수와 통신해야 하며, 마이그레이션된 서비스 도메인 모델 및 의미는 원래 기능과 다릅니다.
- 두 시스템은 의미 체계가 다르고 데이터를 교환해야 하지만 한 시스템이 다른 시스템과 호환되도록 수정하는 것은 실용적이지 않습니다.
- 빠르고 단순화된 접근 방식을 사용하여 영향을 최소화하면서 한 시스템을 다른 시스템에 맞게 조정해야 합니다.
- 애플리케이션이 외부 시스템과 통신하고 있습니다.

문제 및 고려 사항

- 팀 종속성:시스템의 여러 서비스를 여러 팀이 소유하는 경우 마이그레이션된 서비스의 새로운 도메인 모델 시맨틱으로 인해 호출 시스템이 변경될 수 있습니다. 하지만 팀마다 다른 우선순위가 있을

수 있기 때문에 조정된 방식으로 이러한 변경을 수행하지 못할 수도 있습니다. ACL은 수신자를 분리하고 호출을 새 서비스의 의미와 일치하도록 변환하므로 호출자가 현재 시스템을 변경할 필요가 없습니다.

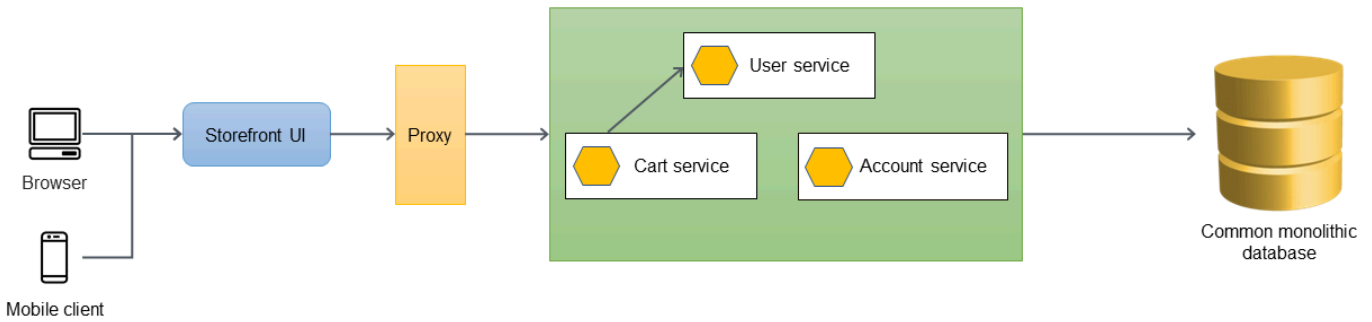
- 운영 오버헤드:ACL 패턴은 운영 및 유지 관리에 추가적인 노력이 필요합니다. 이 작업에는 ACL을 모니터링 및 알림 도구, 릴리스 프로세스, 지속적 통합 및 지속적 전달 (CI/CD) 프로세스와 통합하는 작업이 포함됩니다.
- 단일 장애 지점:ACL에 장애가 발생하면 대상 서비스에 연결할 수 없게 되어 애플리케이션 문제가 발생할 수 있습니다. 이 문제를 완화하려면 재시도 기능과 회로 차단기를 내장해야 합니다. 더 보기 [백 오프로 재시도](#)과 [회로 차단기](#) 패턴을 통해 이러한 옵션에 대해 자세히 알아볼 수 있습니다. 적절한 경고 및 로깅을 설정하면 평균 해결 시간 (MTTR) 이 개선됩니다.
- 기술 부채:마이그레이션 또는 현대화 전략의 일환으로 ACL이 임시 솔루션인지 임시 솔루션인지 아니면 장기 솔루션인지 고려하십시오. 임시 솔루션인 경우 ACL을 기술적 문제로 기록하고 모든 종속 발신자가 마이그레이션된 후에 ACL을 사용 중지해야 합니다.
- 지연 시간:추가 계층으로 인해 한 인터페이스에서 다른 인터페이스로 요청이 변환되어 지연 시간이 발생할 수 있습니다. ACL을 프로덕션 환경에 배포하기 전에 응답 시간에 민감한 애플리케이션의 성능 허용 오차를 정의하고 테스트하는 것이 좋습니다.
- 스케일링 병목 현상:서비스를 최대 부하까지 확장할 수 있는 고부하 애플리케이션에서는 ACL이 병목 현상이 발생하여 확장 문제가 발생할 수 있습니다. 대상 서비스가 필요에 따라 확장되는 경우 그에 따라 확장되도록 ACL을 설계해야 합니다.
- 서비스별 또는 공유 구현:ACL을 공유 객체로 설계하여 호출을 여러 서비스 또는 서비스별 클래스로 변환하고 리디렉션할 수 있습니다. ACL의 구현 유형을 결정할 때 지연 시간, 규모 조정 및 장애 허용 범위를 고려하십시오.

구현

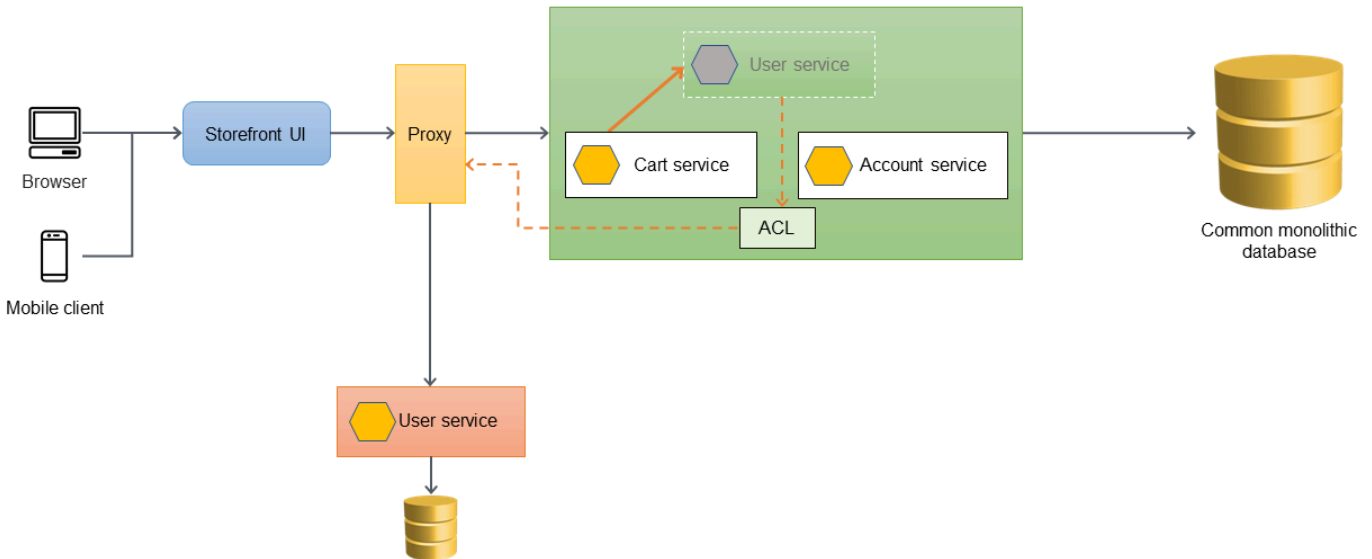
모놀리식 애플리케이션 내에서 ACL을 마이그레이션되는 서비스 전용 클래스로 구현하거나 독립 서비스로 구현할 수 있습니다. 모든 종속 서비스를 마이크로서비스 아키텍처로 마이그레이션한 후에는 ACL을 해제해야 합니다.

고수준 아키텍처

다음 예제 아키텍처에서 모놀리식 애플리케이션에는 사용자 서비스, 카트 서비스, 계정 서비스라는 세 가지 서비스가 있습니다. 카트 서비스는 사용자 서비스에 종속되며 애플리케이션은 모놀리식 관계형 데이터베이스를 사용합니다.

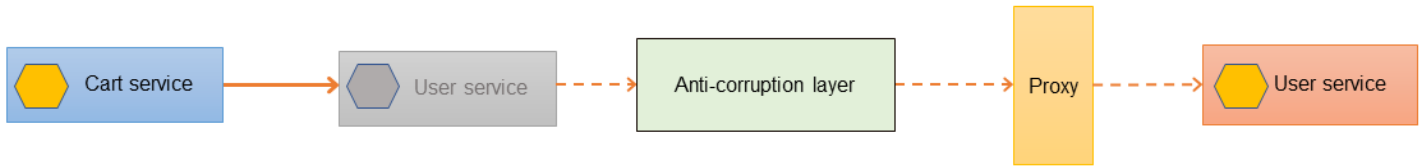


다음 아키텍처에서는 사용자 서비스가 새 마이크로서비스로 마이그레이션되었습니다. 카트 서비스는 사용자 서비스를 호출하지만 모놀리스 내에서는 더 이상 구현을 사용할 수 없습니다. 또한 새로 마이그레이션된 서비스의 인터페이스가 모놀리식 애플리케이션 내에 있었을 때의 이전 인터페이스와 일치하지 않을 수도 있습니다.



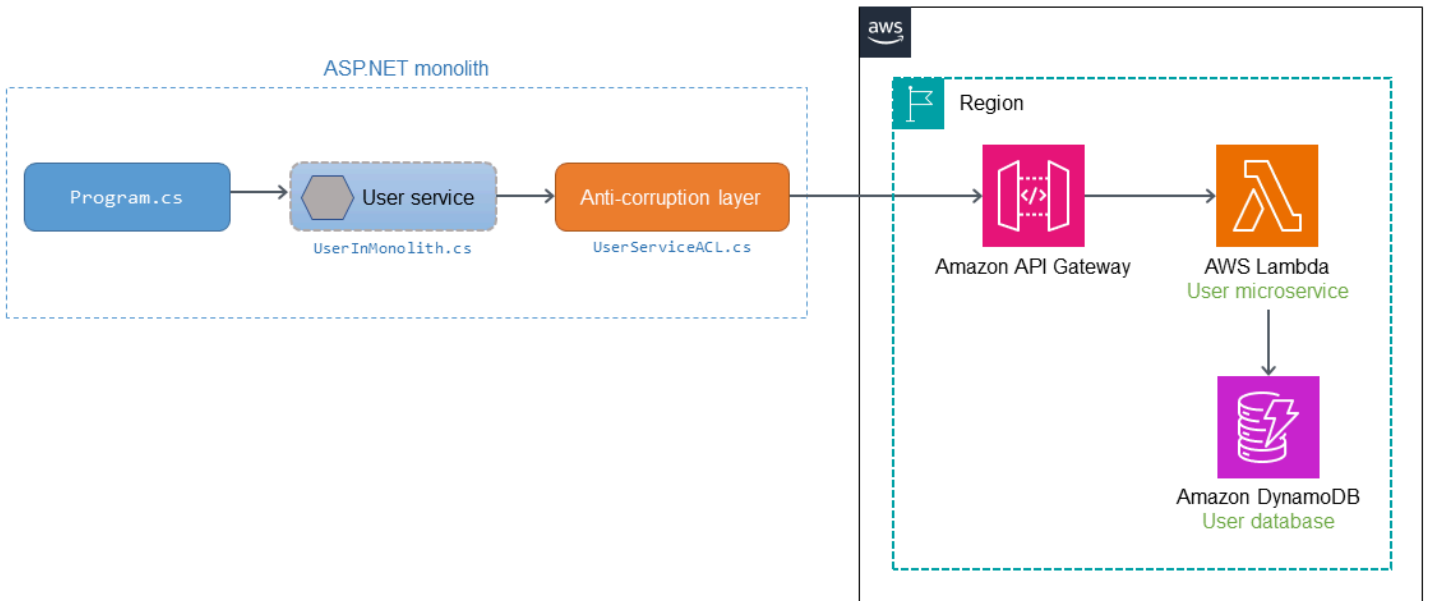
카트 서비스가 새로 마이그레이션된 사용자 서비스를 직접 호출해야 하는 경우 카트 서비스를 변경하고 모놀리식 애플리케이션을 철저히 테스트해야 합니다. 이로 인해 혁신 위험과 비즈니스 중단이 증가할 수 있습니다. 목표는 모놀리식 애플리케이션의 기존 기능에 대한 변경을 최소화하는 것이어야 합니다.

이 경우 이전 사용자 서비스와 새로 마이그레이션된 사용자 서비스 간에 ACL을 도입하는 것이 좋습니다. ACL은 통화를 최신 인터페이스로 변환하는 어댑터 또는 파사드 역할을 합니다. ACL은 모놀리식 애플리케이션 내에서 클래스로 구현될 수 있습니다 (예: UserServiceFacade 또는 UserServiceAdapter) 이는 마이그레이션된 서비스에만 해당됩니다. 모든 종속 서비스를 마이크로서비스 아키텍처로 마이그레이션한 후에는 부패 방지 계층을 해제해야 합니다.



를 사용한 구현AWS서비스

다음 다이어그램은 를 사용하여 이 ACL 예제를 구현하는 방법을 보여줍니다.AWS서비스.



사용자 마이크로서비스는 ASP.NET 모놀리식 애플리케이션에서 마이그레이션되어 다음과 같이 배포됩니다.[AWS Lambda](#)에서 작동합니다. Lambda 함수에 대한 호출은 다음을 통해 라우팅됩니다.[아마존 API 게이트웨이](#). ACL은 모놀리스에 배포되어 호출을 변환하여 사용자 마이크로서비스의 의미에 맞게 조정됩니다.

언제 `Program.cs` 사용자 서비스를 호출합니다 (`UserInMonolith.cs`) 모놀리스 내부에서 통화가 ACL로 라우팅됩니다 (`UserServiceACL.cs`). ACL은 호출을 새로운 시맨틱 및 인터페이스로 변환하고 API Gateway 엔드포인트를 통해 마이크로서비스를 호출합니다. 발신자 (`Program.cs`) 는 사용자 서비스 및 ACL에서 이루어지는 변환 및 라우팅을 인식하지 못합니다. 발신자가 코드 변경을 인식하지 못하기 때문에 비즈니스 중단이 줄어들고 혁신 위험이 줄어듭니다.

샘플 코드

다음 코드 스니펫은 원래 서비스에 대한 변경 사항과 구현을 제공합니다. `UserServiceACL.cs`. 요청이 수신되면 원래 사용자 서비스가 ACL을 호출합니다. ACL은 소스 객체를 새로 마이그레이션된 서비스의 인터페이스와 일치하도록 변환하고, 서비스를 호출하고, 호출자에게 응답을 반환합니다.

```
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;
    }
}
```

GitHub저장소

이 패턴에 대한 샘플 아키텍처의 전체 구현은 다음을 참조하십시오. GitHub리포지토리 주소 <https://github.com/aws-samples/anti-corruption-layer-pattern>.

관련 콘텐츠

- [스트랭글러 무화과 패턴](#)
- [서킷 브레이커 패턴](#)
- [백오프 패턴으로 재시도](#)

API 라우팅 패턴

애자일 개발 환경에서는 자율적인 팀(예: 스퀘드 및 트라이브)이 여러 마이크로서비스를 포함하는 하나 이상의 서비스를 담당합니다. 팀은 소비자가 서비스 및 작업 그룹과 상호 작용할 수 있도록 이러한 서비스를 API로 노출합니다.

호스트 이름과 경로를 사용하여 업스트림 소비자에게 HTTP API를 노출하는 세 가지 주요 방법이 있습니다.

방법	설명	예
호스트 이름 라우팅	각 서비스를 호스트 이름으로 노출합니다.	billing.api.example.com
경로 라우팅	각 서비스를 경로로 노출합니다.	api.example.com/billing
헤더 기반 라우팅	각 서비스를 HTTP 헤더로 노출합니다.	x-example-action: something

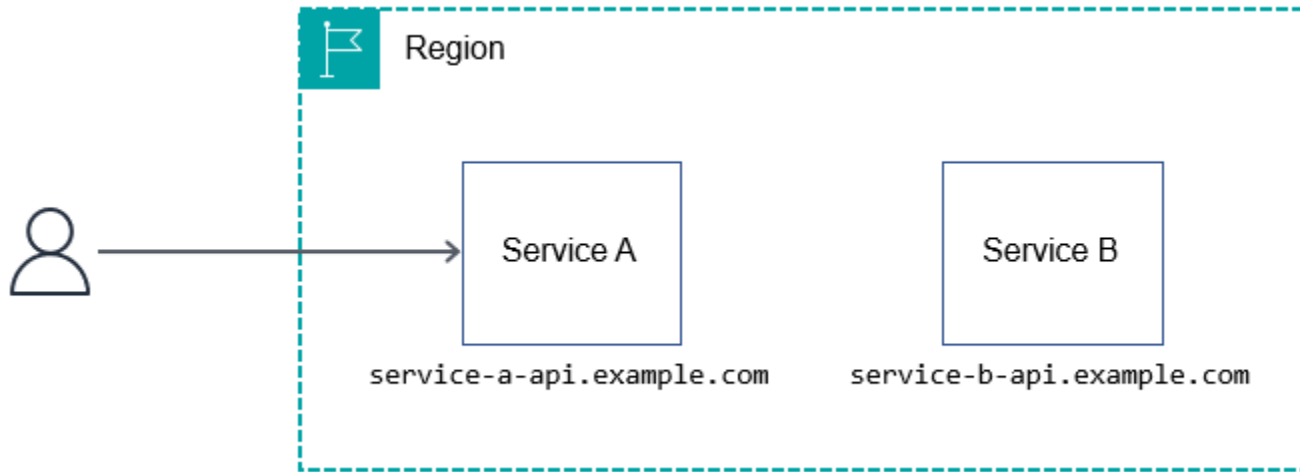
이 섹션에서는 요구 사항 및 조직 구조에 가장 적합한 방법을 결정하는 데 도움이 되도록 이러한 세 가지 라우팅 방법의 일반적인 사용 사례와 장단점을 간략하게 설명합니다.

호스트 이름 라우팅 패턴

호스트 이름별 라우팅은 각 API에 고유한 호스트 이름(예: service-a.api.example.com 또는 service-a.example.com)을 지정하여 API 서비스를 격리하는 메커니즘입니다.

일반적인 사용 사례

호스트 이름을 사용하여 라우팅하면 서비스 팀 간에 아무 것도 공유되지 않기 때문에 릴리스에서 발생하는 마찰이 줄어듭니다. 팀은 DNS 입력부터 프로덕션 환경의 서비스 운영에 이르기까지 모든 것을 관리할 책임이 있습니다.



장점

호스트 이름 라우팅은 HTTP API 라우팅을 위한 가장 간단하고 확장성이 뛰어난 방법입니다. 관련 AWS 서비스를 사용하여 이 방법을 따르는 아키텍처를 구축할 수 있습니다. [Amazon API Gateway](#), [애플리케이션 로드 밸런서](#), [AWS AppSync](#), [Amazon Elastic Compute Cloud \(Amazon EC2\)](#) 또는 기타 [HTTP 호환 서비스를 사용하여](#) 아키텍처를 생성할 수 있습니다.

팀은 호스트 이름 라우팅을 사용하여 하위 도메인을 완전히 소유할 수 있습니다. 또한 특정 버전 또는 버전 (예: 또는) 에 대한 배포를 더 쉽게 분리, 테스트 및 조정할 수 있습니다. AWS 리전 `region.service-a.api.example.com` `dev.region.service-a.api.example.com`

단점

호스트 이름 라우팅을 사용하는 경우 소비자는 노출하는 각 API와 상호 작용할 때 여러 호스트 이름을 기억해야 합니다. 클라이언트 SDK를 제공하여 이 문제를 해결할 수 있습니다. 하지만 클라이언트 SDK에는 고유한 문제가 있습니다. 일례로, 롤링 업데이트, 다국어, 버전 관리, 보안 문제 또는 버그 수정으로 인한 주요 변경 사항 전달, 설명서 등을 지원해야 합니다.

호스트 이름 라우팅을 사용하는 경우 새 서비스를 만들 때마다 하위 도메인이나 도메인도 등록해야 합니다.

경로 라우팅 패턴

경로별 라우팅은 여러 API 또는 모든 API를 동일한 호스트 이름으로 그룹화하고 요청 URI를 사용하여 서비스를 격리하는 메커니즘입니다(예: `api.example.com/service-a` 또는 `api.example.com/service-b`).

일반적인 사용 사례

대부분의 팀은 아키텍처를 단순하게 유지하기 위해 이 방식을 선택합니다. 예를 들어 개발자는 HTTP API와 상호 작용할 때 `api.example.com`과 같은 URL을 하나만 기억하면 됩니다. API 설명서는 여러 포털이나 PDF로 분할되지 않고 함께 보관되는 경우가 많기 때문에 보통 이해하기가 더 쉽습니다.

경로 기반 라우팅은 HTTP API 공유를 위한 간단한 메커니즘으로 간주됩니다. 하지만 구성, 권한 부여, 통합, 다중 흡으로 인한 추가 지연 시간 등의 운영 오버헤드가 수반됩니다. 또한 잘못된 구성으로 인해 모든 서비스가 중단되지 않도록 하려면 완성도 높은 변경 관리 프로세스가 필요합니다.

에서는 AWS 여러 가지 방법으로 API를 공유하고 올바른 서비스로 효과적으로 라우팅할 수 있습니다. 다음 섹션에서는 HTTP 서비스 역방향 프록시, API Gateway 및 Amazon이라는 세 가지 접근 방식에 대해 설명합니다 CloudFront. API 서비스를 통합하기 위한 방법으로 제안된 접근 방식 중 AWS에서 실행 중인 다운스트림 서비스에 의존하는 방식은 없습니다. 서비스는 HTTP와 호환되는 한 문제 없이 또는 어떤 기술로도 어디서든 실행될 수 있습니다.

HTTP 서비스 역방향 프록시

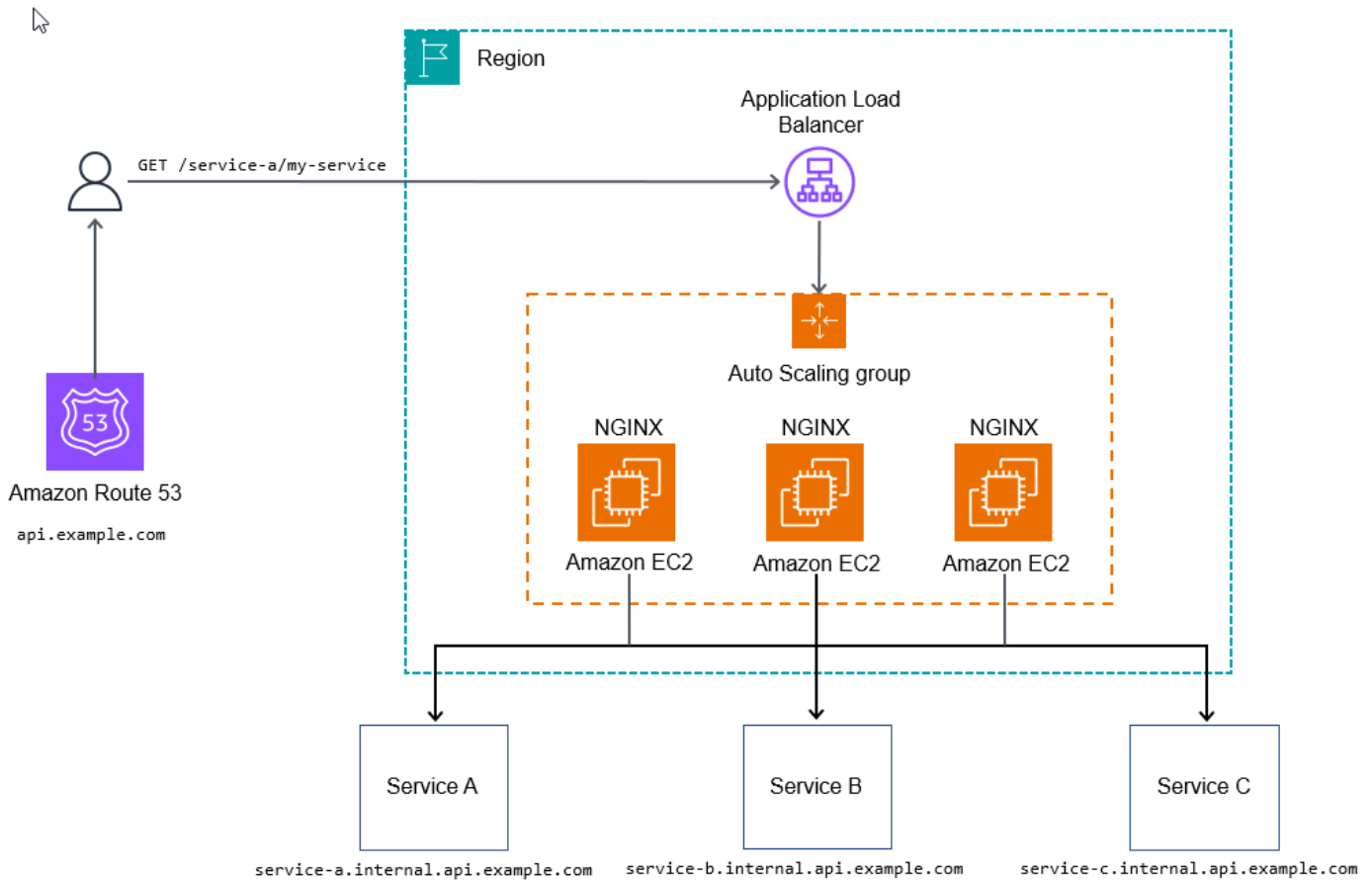
[NGINX](#)와 같은 HTTP 서버를 사용하여 동적 라우팅 구성을 생성할 수 있습니다. [Kubernetes](#) 아키텍처에서는 경로를 서비스와 매칭하는 수신 규칙을 생성할 수도 있습니다. (이 가이드에서는 Kubernetes 수신에 대해 다루지 않습니다. 자세한 내용은 [Kubernetes 설명서](#)를 참조하세요.)

NGINX의 다음 구성은 `api.example.com/my-service/`의 HTTP 요청을 `my-service.internal.api.example.com`에 동적으로 매핑합니다.

```
server {
    listen 80;

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

다음 다이어그램은 HTTP 서비스 역방향 프록시 방법을 보여 줍니다.



요청 처리를 시작하는 데 추가 구성을 사용하지 않고 다운스트림 API가 지표와 로그를 수집할 수 있도록 하는 일부 사용 사례에는 이 접근 방식을 사용해도 충분할 수 있습니다.

운영 프로덕션 준비를 갖추려면 스택의 모든 수준에 관찰성을 추가하거나, 추가 구성을 추가하거나, 속도 제한 또는 사용량 토큰과 같은 고급 기능을 허용하도록 API 수신 지점을 사용자 지정하는 스크립트를 추가해야 합니다.

장점

HTTP 서비스 역방향 프록시 방식의 궁극적인 목표는 API를 단일 도메인으로 통합하여 모든 API 소비자에게 일관되게 보이도록 확장 가능하고 관리하기 쉬운 접근 방식을 만드는 것입니다. 또한 이 접근 방식을 통해 서비스 팀은 배포 후 오버헤드를 최소화하면서 자체 API를 배포하고 관리할 수 있습니다. [AWS X-Ray](#)[AWS WAF](#) 또는와 같은 추적 관리 서비스는 여기에도 여전히 적용됩니다.

단점

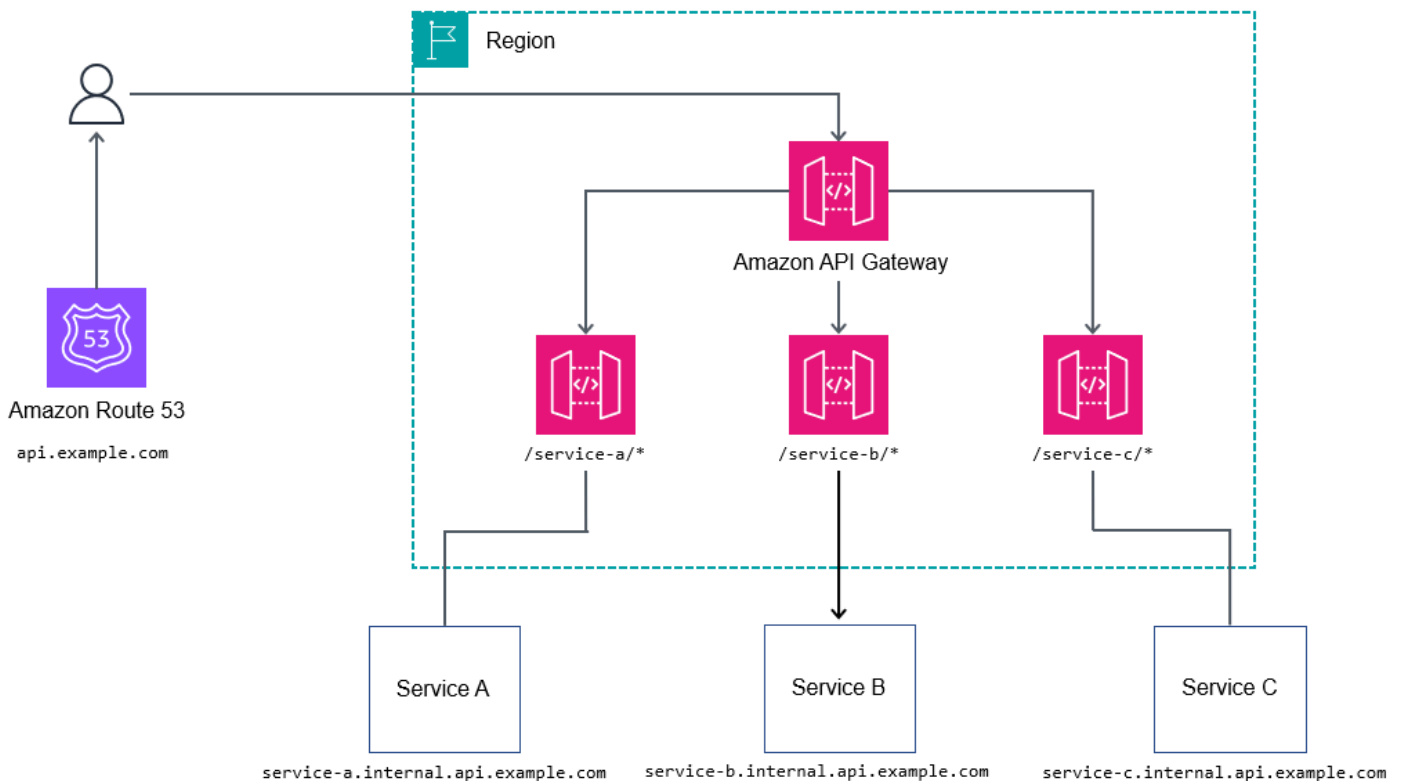
이 접근 방식의 가장 큰 단점은 필요한 인프라 구성 요소를 테스트하고 관리하는 작업 부담이 크다는 것이지만, 사이트 신뢰성 엔지니어링(SRE) 팀이 있으면 문제가 되지 않습니다.

이 방법에는 비용 한계점이 있습니다. 작은 볼륨에서 중간 볼륨까지는 이 가이드에서 설명하는 다른 방법보다 비용이 많이 듭니다. 하지만 큰 볼륨에서는 매우 비용 효율적입니다(초당 트랜잭션 약 10만 건 이상).

API Gateway

[Amazon API Gateway](#) 서비스(REST API 및 HTTP API)는 HTTP 서비스 역방향 프록시 방법과 유사한 방식으로 트래픽을 라우팅할 수 있습니다. HTTP 프록시 모드에서 API Gateway를 사용하면 여러 서비스를 최상위 하위 도메인 `api.example.com`의 진입점으로 래핑한 다음 요청을 중첩 서비스로 프록시할 수 있는 간단한 방법이 제공됩니다(예: `billing.internal.api.example.com`).

루트 또는 코어 API Gateway에 있는 모든 서비스의 모든 경로를 매핑하여 지나치게 세분화하는 것은 바람직하지 않습니다. 대신 `/billing/*`과 같은 와일드카드 경로를 사용하여 요청을 결제 서비스에 전달합니다. 루트 또는 코어 API Gateway의 경로를 모두 매핑하지는 않으므로, API가 변경될 때마다 루트 API 게이트웨이를 업데이트할 필요가 없어 API의 유연성이 향상됩니다.



장점

요청 속성 변경과 같은 더 복잡한 워크플로를 제어하기 위해 REST API는 Apache Velocity Template Language(VTL)를 노출하여 사용자가 요청 및 응답을 수정할 수 있도록 하고 있습니다. REST API는 다음과 같은 추가 이점을 제공할 수 있습니다.

- [AWS Identity and Access Management\(IAM\), 아마존 코그니토 또는 권한 부여자를 통한 인증 N/Z AWS Lambda](#)
- [AWS X-Ray 추적용](#)
- [통합: AWS WAF](#)
- [기본 속도 제한](#)
- 소비자를 여러 티어로 버킷팅하기 위한 사용량 토큰(API Gateway 설명서에서 [처리량 향상을 위해 API 요청 조절](#) 참조)

단점

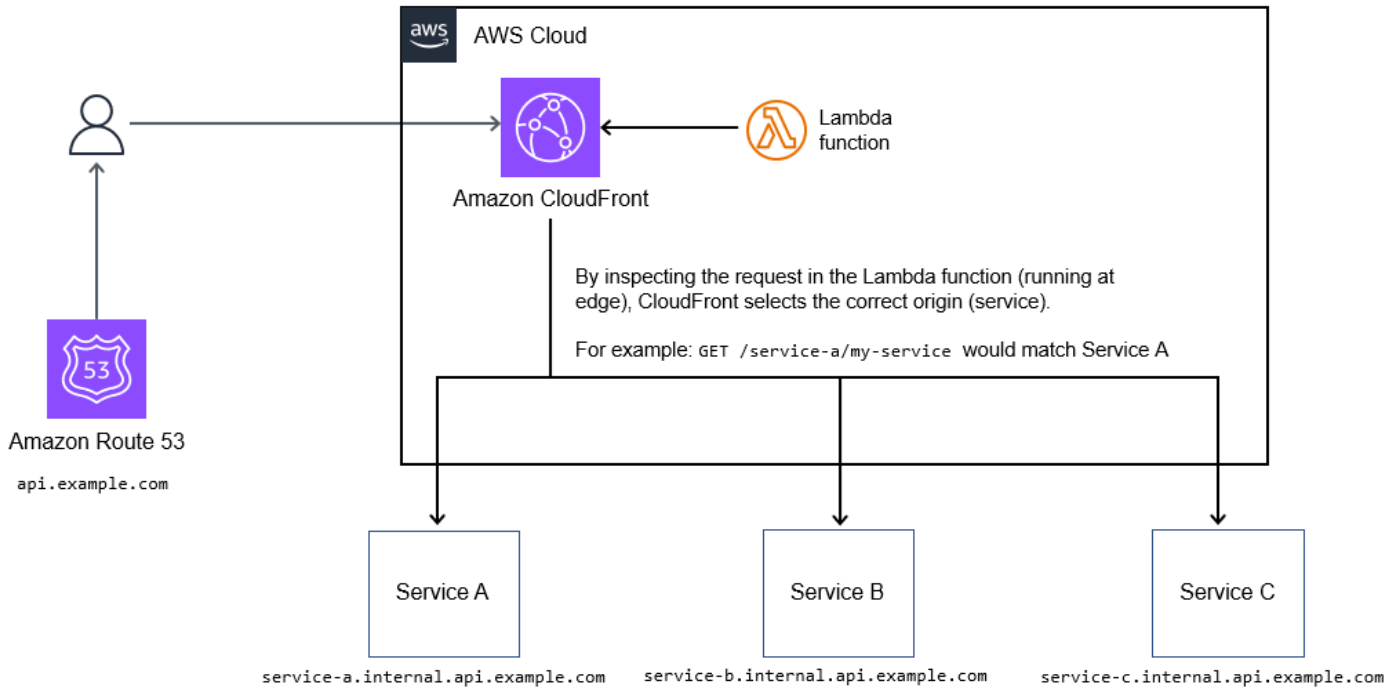
볼륨이 큰 경우 일부 사용자에게는 비용이 문제가 될 수 있습니다.

CloudFront

[CloudFrontAmazon의 동적 오리진 선택 기능을](#) 사용하여 요청을 전달할 오리진 (서비스) 을 조건부로 선택할 수 있습니다. 이 기능을 사용하면 `api.example.com`과 같은 단일 호스트 이름을 통해 여러 서비스를 라우팅할 수 있습니다.

일반적인 사용 사례

라우팅 로직은 Lambda@Edge 함수 내에 코드로 존재하므로, A/B 테스트, canary 릴리스, 기능 플래깅, 경로 재작성과 같은 고도로 맞춤화 가능한 라우팅 메커니즘을 지원합니다. 다음 다이어그램에 이 내용이 잘 설명되어 있습니다.



장점

API 응답 캐싱이 필요한 경우 이 방법은 서비스 컬렉션을 단일 엔드포인트로 통합하는 좋은 방법이 됩니다. API 컬렉션을 통합하는 비용 효율적인 방법입니다.

또한 필드 레벨 암호화는 물론 기본 속도 제한 및 기본 AWS WAF ACL과의 통합을 CloudFront 지원합니다.

단점

이 방법은 통합 가능한 최대 250개의 오리진(서비스)을 지원합니다. 이 한도는 대부분의 배포 사례에서 충분하지만 서비스 포트폴리오를 확장함에 따라 많은 수의 API로 인해 문제가 발생할 수 있습니다.

Lambda @Edge 함수를 업데이트하는 데 현재 몇 분 정도 걸립니다. CloudFront 또한 모든 접속 지점에 변경 사항 전파를 완료하는 데 최대 30분이 소요됩니다. 따라서 업데이트가 완료될 때까지 추가 업데이트가 차단됩니다.

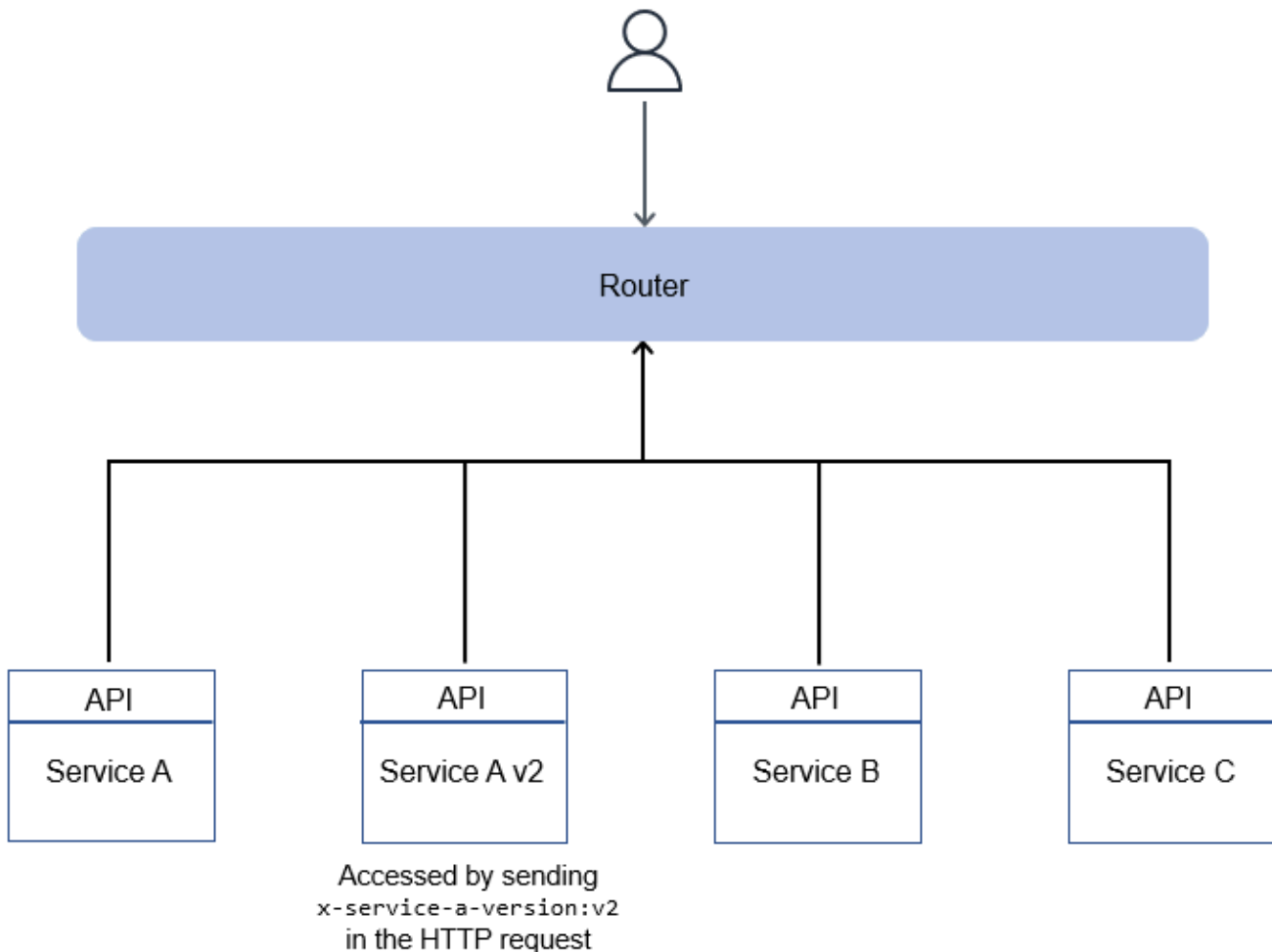
HTTP 헤더 라우팅 패턴

헤더 기반 라우팅을 사용하면 HTTP 요청에 HTTP 헤더를 지정하여 각 요청별로 적절한 서비스를 타겟팅할 수 있습니다. 예를 들어 x-service-a-action: get-thing 헤더를 보내면 Service A에서

get thing을 실행할 수 있습니다. 이 경우에도 요청 경로는 작업하려는 리소스에 대한 지침을 제공하기 때문에 중요합니다.

작업에 HTTP 헤더 라우팅을 사용하는 것 외에, 버전 라우팅, 기능 플래그 활성화, A/B 테스트 또는 이와 유사한 요구 사항을 위한 메커니즘으로 HTTP 헤더 라우팅을 사용할 수도 있습니다. 실제로는 강력한 API를 만들기 위해 헤더 라우팅을 다른 라우팅 방법 중 하나와 함께 사용하게 될 가능성이 높습니다.

HTTP 헤더 라우팅 아키텍처에서는 일반적으로 다음 다이어그램과 같이 올바른 서비스로 라우팅하고 응답을 반환하는 마이크로서비스 앞에 단순한 라우팅 계층을 둡니다. 이 라우팅 계층은 모든 서비스를 포함하거나 버전 기반 라우팅과 같은 작업을 지원하는 일부 서비스만 포함할 수 있습니다.



장점

구성 변경은 최소한의 작업으로 이루어지며 쉽게 자동화할 수 있습니다. 또한 이 방법은 유연하며 서비스에서 원하는 특정 작업만 노출할 수 있는 창의적인 방법을 지원합니다.

단점

호스트 이름 라우팅 방법과 마찬가지로, HTTP 헤더 라우팅은 사용자가 클라이언트를 완전히 제어할 수 있고 사용자 지정 HTTP 헤더를 조작할 수 있다고 가정합니다. 프록시, 콘텐츠 배포 네트워크(CDN), 로드 밸런서가 헤더 크기를 제한할 수 있습니다. 이 제한이 문제가 되는 경우는 드물지만, 추가하는 헤더와 쿠키의 수에 따라 문제가 될 수도 있습니다.

회로 차단기 패턴

의도

회로 차단기 패턴을 사용하면 이전에 반복적으로 시간 초과나 장애가 발생한 경우 발신자 서비스가 다른 서비스 (수신자) 로 통화를 재시도하는 것을 방지할 수 있습니다. 또한 이 패턴은 수신자 서비스가 언제 다시 작동하는지를 감지하는 데에도 사용됩니다.

목적

여러 마이크로서비스가 협업하여 요청을 처리하면 하나 이상의 서비스를 사용할 수 없게 되거나 지연 시간이 길어질 수 있습니다. 복잡한 애플리케이션이 마이크로서비스를 사용하는 경우 한 마이크로서비스가 중단되면 애플리케이션 장애가 발생할 수 있습니다. 마이크로서비스는 원격 프로시저 호출을 통해 통신하므로 네트워크 연결에서 일시적인 오류가 발생하여 장애가 발생할 수 있습니다. (일시적 오류는 백오프 패턴을 [사용한 재시도](#)를 사용하여 처리할 수 있습니다.) 동기 실행 중에 연속적인 시간 초과나 실패로 인해 사용자 경험이 저하될 수 있습니다.

그러나 일부 상황에서는 장애를 해결하는 데 시간이 더 오래 걸릴 수 있습니다. 예를 들어 수신자 서비스가 중단되거나 데이터베이스 경합으로 인해 시간 초과가 발생하는 경우 등이 있습니다. 이러한 경우 호출 서비스가 호출을 반복적으로 재시도하면 이러한 재시도로 인해 네트워크 경합이 발생하고 데이터베이스 스레드 풀이 소비될 수 있습니다. 또한 여러 사용자가 응용 프로그램을 반복해서 다시 시도하면 문제가 악화되고 전체 응용 프로그램의 성능이 저하될 수 있습니다.

회로 차단기 패턴은 마이클 니가르드가 그의 저서 *Release It* (Nygard 2018) 에서 널리 퍼뜨린 것입니다. 이 디자인 패턴은 발신자 서비스가 이전에 반복적으로 시간 초과나 실패를 일으킨 서비스 호출을 재시도하는 것을 방지할 수 있습니다. 또한 수신자 서비스가 언제 다시 작동하는지 감지할 수 있습니다.

회로 차단기 개체는 회로에 이상이 있을 때 자동으로 전류를 차단하는 전기 회로 차단기처럼 작동합니다. 전기 회로 차단기는 고장 발생 시 전류 흐름을 차단하거나 끊어냅니다. 마찬가지로 회로 차단기 오브젝트는 발신자와 수신자 서비스 사이에 위치하며 수신자를 사용할 수 없는 경우 트립됩니다.

[분산 컴퓨팅의 오류는 피터 도이치](#) (Peter Deutsch) 와 Sun Microsystems의 다른 직원들이 주장한 일련의 주장입니다. 분산 응용 프로그램을 처음 접하는 프로그래머는 항상 잘못된 가정을 한다고 합니다. 네트워크 안정성, 제로 레이턴시 기대, 대역폭 제한으로 인해 소프트웨어 애플리케이션은 네트워크 오류에 대한 최소한의 오류 처리로 작성됩니다.

네트워크가 중단되는 동안 애플리케이션은 응답을 무기한으로 기다려 애플리케이션 리소스를 지속적으로 소비할 수 있습니다. 네트워크를 사용할 수 있게 되었을 때 작업을 재시도하지 않으면 응용 프로그램 성능이 저하될 수도 있습니다. 네트워크 문제로 인해 데이터베이스 또는 외부 서비스에 대한 API 호출 제한 시간이 초과되면 회로 차단기 없이 반복적으로 호출하면 비용과 성능에 영향을 미칠 수 있습니다.

적용 가능성

다음과 같은 경우에 이 패턴을 사용하십시오.

- 발신자 서비스가 전화를 걸지만 실패할 가능성이 높습니다.
- 수신자 서비스의 지연 시간이 길면 (예: 데이터베이스 연결 속도가 느린 경우) 수신자 서비스에 대한 제한 시간이 초과됩니다.
- 발신자 서비스가 동기적으로 전화를 걸지만 수신자 서비스를 사용할 수 없거나 대기 시간이 깁니다.

문제 및 고려 사항

- 서비스에 구매받지 않는 구현: 코드 팽창을 방지하려면 마이크로서비스에 구매받지 않고 마이크로 서비스에 구매받지 않는 방식으로 회로 차단기 개체를 구현하는 것이 좋습니다. API
- 수신자별 회로 폐쇄: 피호출자가 성능 문제 또는 장애에서 복구되면 회로 상태를 로 업데이트할 수 있습니다. CLOSED 이는 회로 차단기 패턴의 확장으로, 복구 시간 목표 () RTO 에서 요구하는 경우 구현할 수 있습니다.
- 멀티스레드 통화: 만료 제한 시간 값은 서비스 가용성을 확인하기 위해 통화가 다시 라우팅되기 전까지 회로가 트립된 상태로 유지되는 기간으로 정의됩니다. 피호출자 서비스가 여러 스레드에서 호출되는 경우, 실패한 첫 번째 호출이 만료 타임아웃 값을 정의합니다. 구현시 후속 호출로 인해 만료 제한 시간이 끝없이 바뀌지 않도록 해야 합니다.
- 회로 강제 개설 또는 폐쇄: 시스템 관리자는 회로를 열거나 닫을 수 있어야 합니다. 데이터베이스 테이블의 만료 제한 시간 값을 업데이트하여 이 작업을 수행할 수 있습니다.
- 관찰 가능성: 회로 차단기가 열려 있을 때 실패하는 호출을 식별할 수 있도록 응용 프로그램에 로깅을 설정해야 합니다.

구현

전반적인 아키텍처

다음 예제에서 발신자는 주문 서비스이고 수신자는 결제 서비스입니다.

장애가 없는 경우 주문 서비스는 다음 다이어그램과 같이 회로 차단기를 통해 모든 통화를 결제 서비스로 라우팅합니다.



결제 서비스 제한 시간이 초과되면 회로 차단기가 시간 초과를 감지하고 장애를 추적할 수 있습니다.



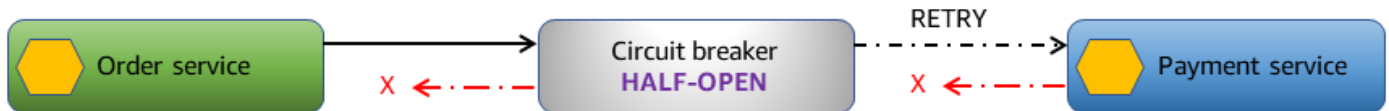
Circuit breaker with payment service failure

제한 시간이 지정된 임계값을 초과하면 응용 프로그램이 회로를 엽니다. 회로가 열려 있는 경우 회로 차단기 객체는 통화를 결제 서비스로 라우팅하지 않습니다. 주문 서비스가 결제 서비스에 전화를 걸면 즉시 실패를 반환합니다.



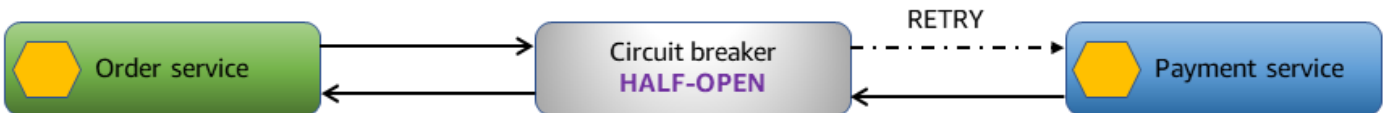
Circuit breaker stops routing to payment service

서킷 브레이커 객체는 주기적으로 결제 서비스에 대한 호출이 성공했는지 확인하려고 합니다.



Circuit breaker periodically retries payment service

결제 서비스 호출이 성공하면 회로가 닫히고 이후의 모든 통화는 결제 서비스로 다시 라우팅됩니다.



Circuit breaker with working payment service

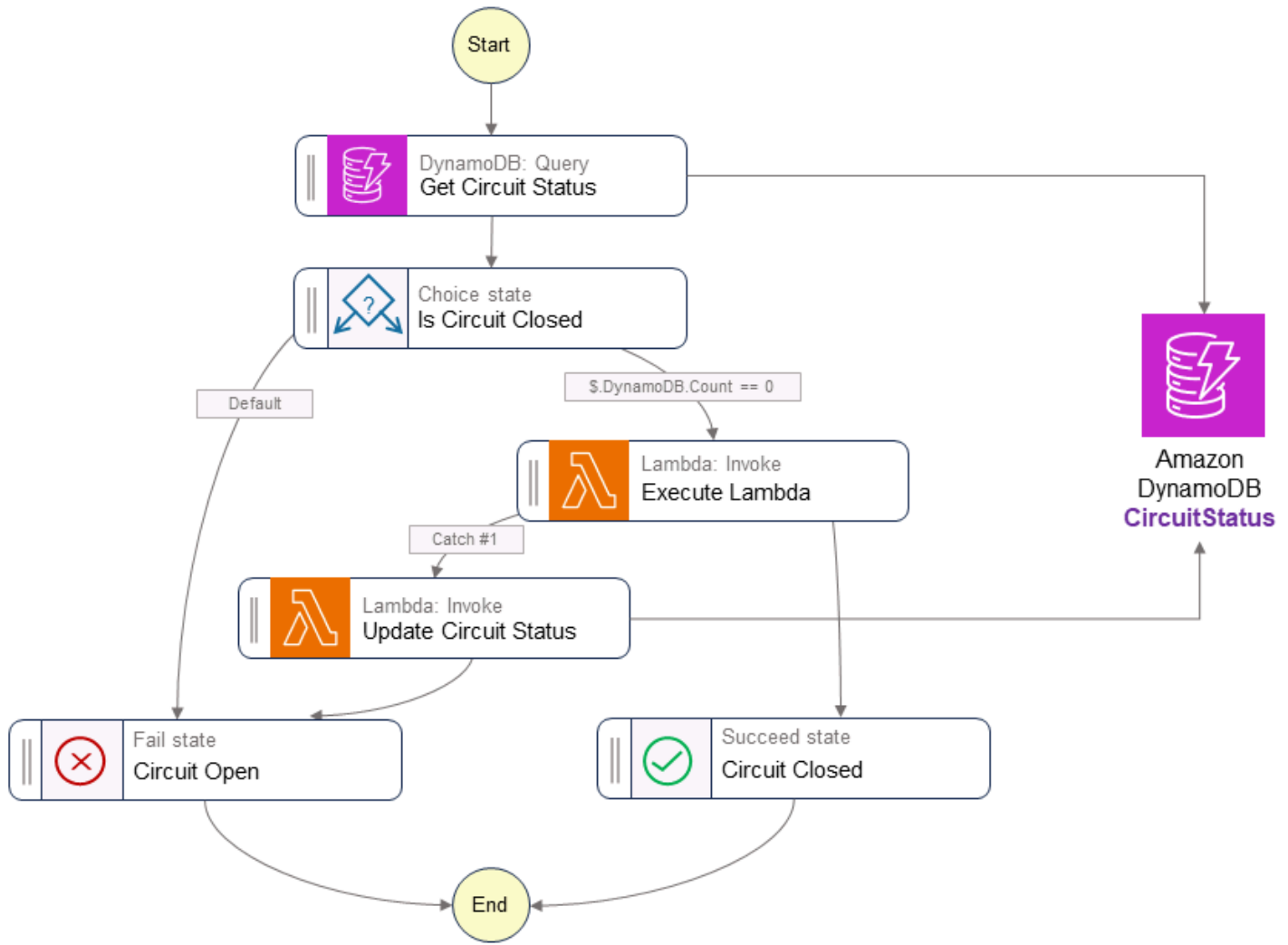
서비스를 이용한 AWS 구현

샘플 솔루션은 고속 워크플로를 사용하여 회로 차단기 패턴을 구현합니다. [AWS Step Functions](#) Step Functions 상태 머신을 사용하면 패턴 구현에 필요한 재시도 기능 및 의사 결정 기반 제어 흐름을 구성할 수 있습니다.

또한 이 솔루션은 [Amazon DynamoDB](#) 테이블을 데이터 스토어로 사용하여 회로 상태를 추적합니다. 성능 향상을 위해 [Amazon ElastiCache \(RedisOSS\)](#) 과 같은 인메모리 데이터스토어로 대체할 수 있습니다.

서비스가 다른 서비스를 호출하려는 경우 수신자 서비스의 이름으로 워크플로를 시작합니다. 워크플로는 현재 성능이 저하된 서비스를 저장하는 CircuitStatus DynamoDB 테이블에서 회로 차단기 상태를 가져옵니다. 피호출자에 대한 만료되지 않은 레코드가 CircuitStatus 포함된 경우 회로는 열려 있습니다. Step Functions 워크플로는 즉각적인 실패를 반환하고 FAIL 상태와 함께 종료됩니다.

CircuitStatus테이블에 피호출자에 대한 레코드가 없거나 만료된 레코드가 포함되어 있는 경우 서비스는 작동 가능합니다. 상태 머신 정의의 ExecuteLambda 단계는 파라미터 값을 통해 전송된 Lambda 함수를 호출합니다. 호출이 성공하면 Step Functions 워크플로가 상태와 함께 종료됩니다. SUCCESS



서비스 호출이 실패하거나 타임아웃이 발생하는 경우 애플리케이션은 정의된 횟수만큼 지수 백오프를 사용하여 재시도합니다. 재시도 후 서비스 호출이 실패하면 워크플로는 an이 있는 서비스의 CircuitStatus 테이블에 레코드를 삽입하고 워크플로우는 상태와 함께 ExpiryTimeStamp 종료됩니다. FAIL 이후에 동일한 서비스를 호출하면 회로 차단기가 열려 있는 한 즉시 오류가 발생합니다. 스테이트 머신 정의의 Get Circuit Status 단계에서는 ExpiryTimeStamp 값을 기반으로 서비스 가용성을 확인합니다. 만료된 항목은 DynamoDB timeto live () 기능을 사용하여 CircuitStatus 테이블에서 삭제됩니다. TTL

샘플 코드

다음 코드는 GetCircuitStatus Lambda 함수를 사용하여 회로 차단기 상태를 확인합니다.

```

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 = "";
}
```

다음 코드는 Step Functions 워크플로의 Amazon States 언어 명령문을 보여줍니다.

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

GitHub 리포지토리

이 패턴의 샘플 아키텍처를 완전히 구현하려면 의 GitHub 저장소를 참조하십시오 <https://github.com/aws-samples/circuit-breaker-netcore-blog>.

블로그 참조

- [Amazon AWS Step Functions DynamoDB와 함께 회로 차단기 패턴 사용](#)

관련 콘텐츠

- [Strangler Fig 패턴](#)
- [백오프 패턴으로 재시도](#)
- [AWS App Mesh 회로 차단기 기능](#)

이벤트 소싱 패턴

Intent

이벤트 기반 아키텍처에서 이벤트 소싱 패턴은 데이터 스토어의 상태 변경을 초래하는 이벤트를 저장합니다. 따라서 상태 변경의 전체 기록을 캡처 및 유지할 수 있으며, 감사 가능성, 추적 가능성 및 과거 상태를 분석하는 능력이 향상됩니다.

목적

여러 마이크로서비스가 유기적으로 작동하여 요청을 처리할 수 있으며 이벤트를 통해 통신합니다. 이러한 이벤트로 인해 상태(데이터)가 변경될 수 있습니다. 이벤트 객체를 발생 순서대로 저장하면 데이터 엔터티의 현재 상태에 대한 중요한 정보와 어떻게 해당 상태에 도달했는지에 대한 추가 정보가 제공됩니다.

적용 가능성

다음과 같은 경우 이벤트 소싱 패턴을 사용합니다.

- 이벤트를 추적하려면 애플리케이션에서 발생하는 이벤트의 변경 불가능한 기록이 필요합니다.
- 신뢰할 수 있는 단일 소스(SSOT)를 기반으로 한 다국어 데이터 프로젝션이 필요합니다.
- 애플리케이션 상태의 특정 시점 재구성이 필요합니다.
- 애플리케이션 상태를 장기간 저장할 필요는 없지만 필요에 따라 재구성하기를 원합니다.
- 워크로드마다 읽기 및 쓰기 볼륨이 다릅니다. 예를 들어 실시간 처리가 필요하지 않은 쓰기 집약적인 워크로드가 있습니다.
- 애플리케이션 성능 및 기타 지표를 분석하기 위한 변경 데이터 캡처(CDC)가 필요합니다.
- 보고 및 규정 준수를 위해 시스템에서 발생하는 모든 이벤트에 대한 감사 데이터가 필요합니다.
- 재생 프로세스 중에 이벤트를 변경(삽입, 업데이트 또는 삭제)하여 가능한 종료 상태를 파악함으로써 가상(what-if) 시나리오를 도출하고자 합니다.

문제 및 고려 사항

- 낙관적 동시성 제어: 이 패턴은 시스템의 상태 변경을 유발하는 모든 이벤트를 저장합니다. 여러 사용자 또는 서비스가 동시에 동일한 데이터를 업데이트하려고 시도함에 따라 이벤트 충돌이 발생할

수 있습니다. 이러한 충돌은 충돌하는 이벤트가 동시에 생성되고 적용될 때 발생하며, 그 결과 최종 데이터 상태가 실제와 일치하지 않게 됩니다. 이 문제를 해결하기 위해 이벤트 충돌을 감지하고 해결하는 전략을 구현할 수 있습니다. 예를 들어 버전 관리를 포함하거나 이벤트에 타임스탬프를 추가하여 업데이트 순서를 추적함으로써 낙관적 동시성 제어 스키마를 구현할 수 있습니다.

- **복잡성:** 이벤트 소싱을 구현하려면 기존 CRUD 운영에서 이벤트 기반 방식으로 발상을 전환해야 합니다. 시스템을 원래 상태로 복원하는 데 사용되는 재생 프로세스는 데이터 멱등성을 보장하기 위해 복잡하게 구현될 수 있습니다. 게다가 이벤트 스토리지, 백업 및 스냅샷도 복잡성을 가중시킬 수 있습니다.
- **최종 일관성:** 명령 쿼리 책임 분리(CQRS) 패턴 또는 구체화된 뷰를 사용하여 데이터를 업데이트하는 데 따른 지연 시간 덕분에 이벤트에서 파생된 데이터 프로젝션에서는 결국 일관성이 유지됩니다. 소비자가 이벤트 스토어의 데이터를 처리하고 게시자가 새 데이터를 보내는 경우, 데이터 프로젝션 또는 애플리케이션 객체가 현재 상태를 반영하지 않을 수 있습니다.
- **쿼리:** 이벤트 로그에서 현재 또는 집계 데이터를 검색하는 작업은 기존 데이터베이스에 비해 더 복잡하고 느릴 수 있습니다. 특히 복잡한 쿼리 및 보고 작업의 경우 더욱 그렇습니다. 이 문제를 완화하기 위해 이벤트 소싱은 CQRS 패턴으로 구현되는 경우가 많습니다.
- **이벤트 스토어의 크기 및 비용:** 특히 이벤트 처리량이 많거나 보존 기간이 연장된 시스템의 경우, 이벤트가 지속적으로 유지됨에 따라 이벤트 스토어의 크기가 기하급수적으로 증가할 수 있습니다. 따라서 이벤트 스토어가 너무 커지는 것을 방지하기 위해 정기적으로 이벤트 데이터를 비용 효율적인 스토리지에 보관해야 합니다.
- **이벤트 스토어의 확장성:** 이벤트 스토어는 대량의 쓰기 및 읽기 작업을 모두 효율적으로 처리해야 합니다. 이벤트 스토어는 규모를 조정하기가 어려울 수 있으므로, 샤드와 파티션을 제공하는 데이터 스토어를 사용하는 것이 중요합니다.
- **효율성 및 최적화:** 쓰기 및 읽기 작업을 모두 효율적으로 처리하는 이벤트 스토어를 선택하거나 설계합니다. 이벤트 스토어는 애플리케이션의 예상 이벤트 볼륨 및 쿼리 패턴에 맞게 최적화되어야 합니다. 인덱싱 및 쿼리 메커니즘을 구현하면 애플리케이션 상태를 재구성할 때 이벤트 검색 속도를 높일 수 있습니다. 쿼리 최적화 기능을 제공하는 특수한 이벤트 스토어 데이터베이스 또는 라이브러리를 사용하는 것도 고려해 볼 수 있습니다.
- **스냅샷:** 시간 기반 활성화를 통해 정기적으로 이벤트 로그를 백업해야 합니다. 마지막으로 성공한 것으로 알려진 데이터 백업의 이벤트를 재생하면 애플리케이션 상태가 특정 시점으로 복구됩니다. Recovery Point Objective(RPO)는 마지막 데이터 복구 시점 이후 허용되는 최대 시간입니다. RPO는 마지막 복구 시점과 서비스 중단 사이에 허용되는 데이터 손실로 간주되는 범위를 결정합니다. 데이터 및 이벤트 스토어의 일일 스냅샷 빈도는 애플리케이션의 RPO를 기준으로 해야 합니다.
- **시간 민감도:** 이벤트는 발생한 순서대로 저장됩니다. 따라서 이 패턴을 구현할 때 고려해야 할 중요한 요소는 네트워크 신뢰성입니다. 지연 문제로 인해 잘못된 시스템 상태가 발생할 수 있습니다. 최대 한 번 전송되는 선입선출(FIFO) 대기열을 사용하여 이벤트를 이벤트 스토어로 전송하세요.

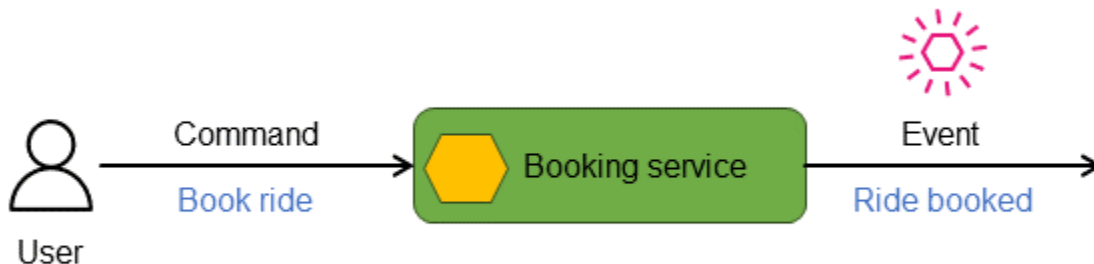
- 이벤트 재생 성능: 많은 수의 이벤트를 재생하여 현재 애플리케이션 상태를 재구성하려면 시간이 많이 걸릴 수 있습니다. 특히 보관된 데이터에서 이벤트를 재생할 때는 성능을 개선하기 위한 최적화 노력이 필요합니다.
- 외부 시스템 업데이트: 이벤트 소싱 패턴을 사용하는 애플리케이션은 외부 시스템의 데이터 스토어를 업데이트하고 이러한 업데이트를 이벤트 객체로 캡처할 수 있습니다. 이벤트 재생 중에, 외부 시스템이 업데이트를 예상하지 않으면 이는 문제가 될 수 있습니다. 이러한 경우 기능 플래그를 사용하여 외부 시스템 업데이트를 제어할 수 있습니다.
- 외부 시스템 쿼리: 외부 시스템 직접 호출이 호출 날짜 및 시간에 민감한 경우, 수신된 데이터를 내부 데이터 스토어에 저장하여 재생 중에 사용할 수 있습니다.
- 이벤트 버전 관리: 애플리케이션이 발전함에 따라 이벤트 구조(스키마)가 변경될 수 있습니다. 이전 버전과 이후 버전의 호환성을 보장하기 위해 이벤트에 대한 버전 관리 전략을 구현해야 합니다. 여기에는 이벤트 페이로드에 version 필드를 포함하고 재생 중에 다양한 이벤트 버전을 적절하게 처리하는 것이 포함될 수 있습니다.

구현

전반적인 아키텍처

명령 및 이벤트

분산된 이벤트 기반 마이크로서비스 애플리케이션에서 명령은 일반적으로 상태 변경을 시작할 목적으로 서비스에 전송되는 명령 또는 요청을 나타냅니다. 서비스는 이러한 명령을 처리하고 명령의 유효성과 현재 상태에 대한 적용 가능성을 평가합니다. 명령이 성공적으로 실행되면 서비스는 수행된 조치와 관련 상태 정보를 나타내는 이벤트를 보내 응답합니다. 예를 들어 다음 다이어그램에서 예약 서비스는 Ride booked 이벤트를 생성하여 Book ride 명령에 응답합니다.



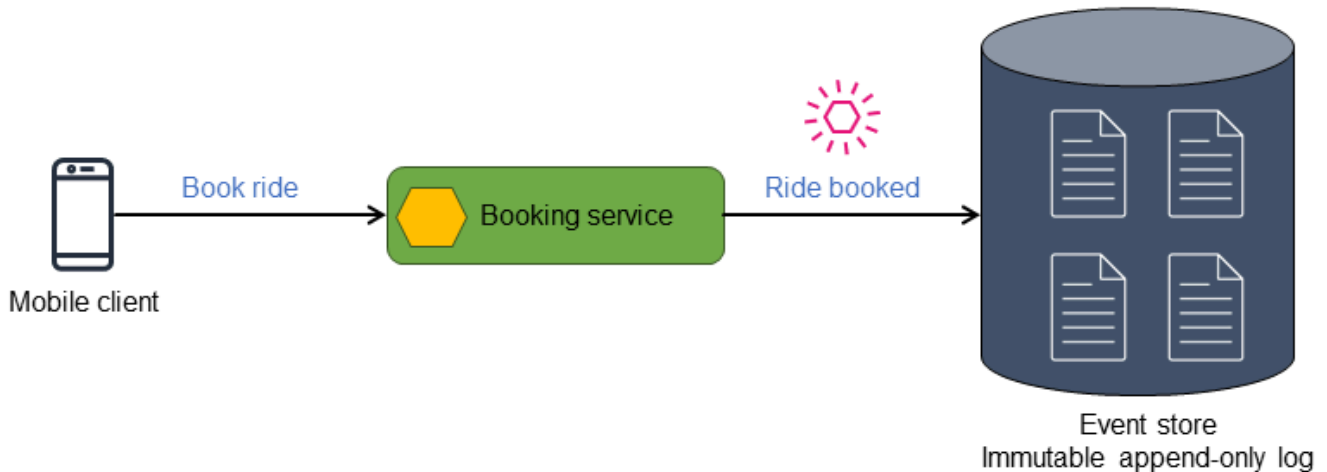
이벤트 스토어

이벤트는 변경 불가능하고 추가만 가능하며 시간순으로 정렬된 리포지토리 또는 이벤트 스토어라는 데이터 스토어에 기록됩니다. 각 상태 변경은 개별 이벤트 객체로 취급됩니다. 이벤트를 발생 순서대로 재생하여 초기 상태, 현재 상태 및 시점 뷰가 알려진 엔터티 객체 또는 데이터 스토어를 재구성할 수 있습니다.

이벤트 스토어는 모든 작업 및 상태 변경에 대한 기록 역할을 하며, 신뢰할 수 있는 단일 소스로서 중요한 역할을 합니다. 이벤트 스토어를 사용하면 재생 프로세서를 통해 이벤트를 전달하여 시스템의 최종, 최신 상태를 도출할 수 있습니다. 재생 프로세서는 이러한 이벤트를 적용하여 최신 시스템 상태를 정확하게 나타냅니다. 또한 이벤트 스토어를 사용하면 재생 프로세서를 통해 이벤트를 재생하여 특정 시점의 상태를 파악할 수 있습니다. 이벤트 소싱 패턴에서는 최신 이벤트 객체가 현재 상태를 완전하게 나타내지 않을 수 있습니다. 다음 세 가지 방법 중 하나로 현재 상태를 도출할 수 있습니다.

- 관련 이벤트를 집계. 관련 이벤트 객체를 결합하여 쿼리를 위한 현재 상태를 생성합니다. 이 방식은 이벤트가 결합되어 읽기 전용 데이터 스토어에 기록된다는 점에서 CQRS 패턴과 함께 사용되는 경우가 많습니다.
- 구체화된 뷰 사용. 구체화된 뷰 패턴과 함께 이벤트 소싱을 사용하여 이벤트 데이터를 계산하거나 요약하고 관련 데이터의 현재 상태를 얻을 수 있습니다.
- 이벤트를 재생. 이벤트 객체를 재생하여 현재 상태를 생성하는 작업을 수행할 수 있습니다.

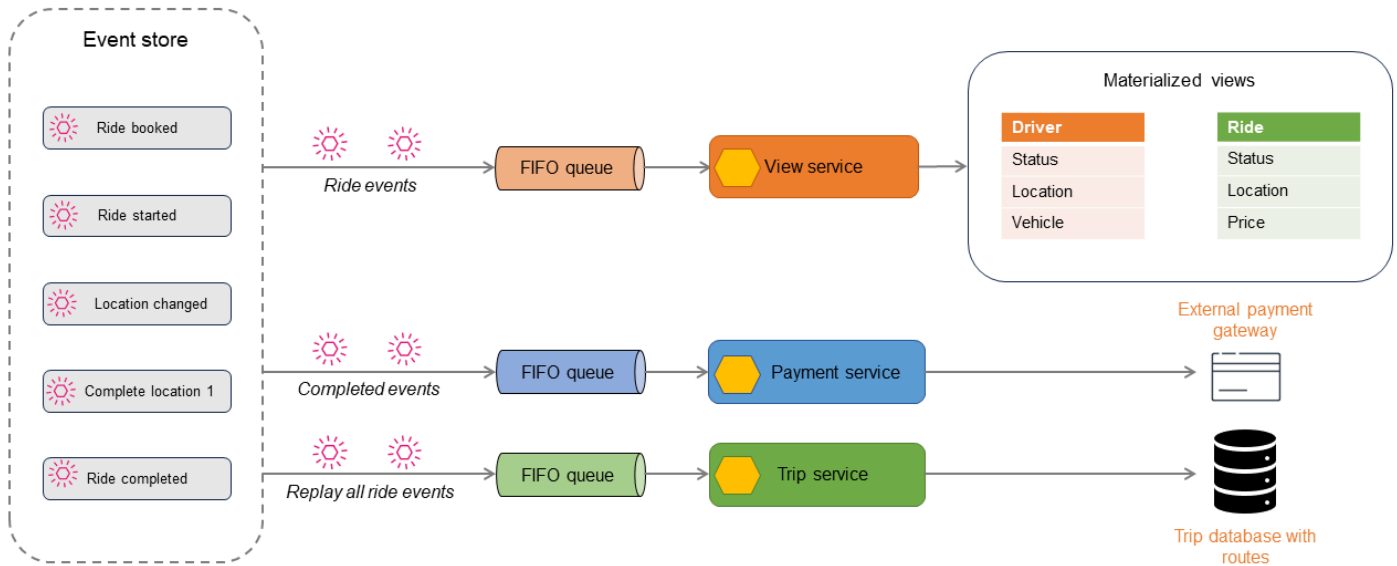
다음 다이어그램은 Ride booked 이벤트 스토어에 저장되는 이벤트를 보여줍니다.



이벤트 스토어는 저장하는 이벤트를 게시하고, 이벤트를 필터링하여 후속 작업을 위해 적절한 프로세서로 라우팅할 수 있습니다. 예를 들어 상태를 요약하고 구체화된 뷰를 보여주는 뷰 프로세서로 이벤트를 라우팅할 수 있습니다. 이벤트는 대상 데이터 스토어의 데이터 형식으로 변환됩니다. 이 아키텍처를

확장하여 다양한 유형의 데이터 스토어를 도출할 수 있으며, 이를 통해 데이터의 다국어 지속성이 보장됩니다.

다음 다이어그램은 차량 예약 애플리케이션의 이벤트를 설명합니다. 애플리케이션 내에서 발생하는 모든 이벤트는 이벤트 스토어에 저장됩니다. 그런 다음 저장된 이벤트가 필터링되어 다른 소비자에게 라우팅됩니다.



주행 이벤트는 CQRS 또는 구체화된 뷰 패턴을 사용하여 읽기 전용 데이터 스토어를 생성하는 데 사용할 수 있습니다. 읽기 스토어를 쿼리하여 차량 서비스, 운전자 또는 예약의 현재 상태를 확인할 수 있습니다. Location changed 또는 Ride completed 같은 일부 이벤트는 결제 처리를 위해 다른 소비자에게 게시됩니다. 주행이 완료되면 모든 주행 이벤트가 재생되어 감사 또는 보고 목적으로 주행 기록이 작성됩니다.

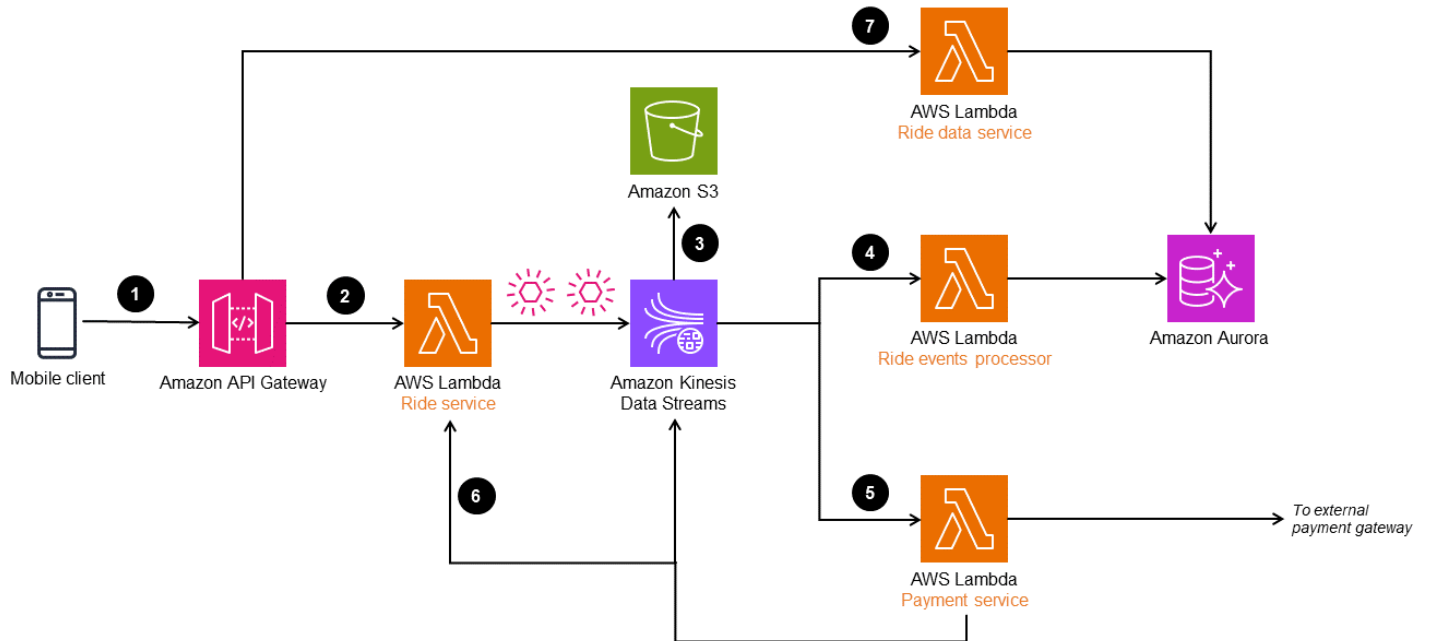
이벤트 소싱 패턴은 시점 복구가 필요한 애플리케이션에서 자주 사용되며, 신뢰할 수 있는 단일 소스를 사용하여 데이터를 다양한 형식으로 프로젝션해야 하는 경우에도 자주 사용됩니다. 이 두 작업 모두 이벤트를 실행하고 필요한 종료 상태를 도출하기 위한 재생 프로세스를 필요로 합니다. 재생 프로세서에는 알려진 시작 지점이 필요할 수도 있습니다. 애플리케이션 시작 시부터 이 프로세스를 시작하는 것은 효율적이지 않습니다. 시스템 상태의 스냅샷을 주기적으로 생성하고 이벤트 수를 줄여 최신 상태를 도출하는 것이 좋습니다.

AWS 서비스를 사용한 구현

다음 아키텍처에서는 Amazon Kinesis Data Streams가 이벤트 스토어로 사용됩니다. 이 서비스는 애플리케이션 변경 사항을 이벤트로 캡처 및 관리하고 처리량이 많은 실시간 데이터 스트리밍 솔루션

을 제공합니다. AWS에서 이벤트 소싱 패턴을 구현하려면 애플리케이션의 요구 사항에 따라 Amazon EventBridge 및 Amazon Managed Streaming for Apache Kafka(Amazon MSK)와 같은 서비스를 사용할 수도 있습니다.

내구성을 강화하고 감사를 활성화하려는 경우에는 Kinesis Data Streams에서 캡처한 이벤트를 Amazon Simple Storage Service(S3)에 보관할 수 있습니다. 이 이중 스토리지 접근 방식은 향후 분석 및 규정 준수를 위해 과거 이벤트 데이터를 안전하게 보존하는 데 도움이 됩니다.



이 워크플로는 다음 단계로 구성됩니다.

1. 모바일 클라이언트를 통해 Amazon API Gateway 엔드포인트에 차량 예약 요청이 전송됩니다.
2. 이 주행 마이크로서비스(Ride service Lambda 함수)는 요청을 수신하고, 객체를 변환하고, Kinesis Data Streams에 게시합니다.
3. Kinesis Data Streams의 이벤트 데이터가 규정 준수 및 감사 기록 목적으로 Amazon S3에 저장됩니다.
4. 이 이벤트는 Ride event processor Lambda 함수에 의해 변환 및 처리되고 Amazon Aurora 데이터베이스에 저장되어 주행 데이터에 대한 구체화된 뷰를 제공합니다.
5. 완료된 주행 이벤트는 필터링되어 결제 처리를 위해 외부 결제 게이트웨이로 전송됩니다. 결제가 완료되면 주행 데이터베이스를 업데이트하기 위해 또 다른 이벤트가 Kinesis Data Streams로 전송됩니다.
6. 주행이 완료되면 주행 이벤트가 Ride service Lambda 함수로 재생되어 경로와 주행 기록을 생성합니다.

7. Aurora 데이터베이스에서 데이터를 읽는 Ride data service를 통해 주행 정보를 읽을 수 있습니다.

또한 API Gateway는 Ride service Lambda 함수를 사용하지 않고 이벤트 객체를 Kinesis Data Streams로 직접 전송할 수 있습니다. 하지만 차량 호출 서비스와 같은 복잡한 시스템에서는 이벤트 객체를 데이터 스트림으로 수집하기 전에 처리하고 보강해야 할 수 있습니다. 이러한 이유로 이 아키텍처에는 이벤트를 Kinesis Data Streams로 보내기 전에 이벤트를 처리하는 Ride service가 있습니다.

블로그 참조

- [New for AWS Lambda – SQS FIFO as an event source](#)

육각형 아키텍처 패턴

의도

포트 및 어댑터 패턴이라고도 하는 육각형 아키텍처 패턴은 2005년에 Alistair Cockburn 박사가 제안한 것입니다. 이는 데이터 저장소나 사용자 인터페이스 (UI) 에 의존하지 않고 애플리케이션 구성 요소를 독립적으로 테스트할 수 있는 느슨하게 결합된 아키텍처를 만드는 것을 목표로 합니다. 이 패턴은 데이터 저장소 및 UI의 기술 종속성을 방지하는 데 도움이 됩니다. 따라서 시간이 지남에 따라 비즈니스 로직에 미치는 영향이 거의 없거나 전혀 없이 기술 스택을 쉽게 변경할 수 있습니다. 이 느슨하게 결합된 아키텍처에서 응용 프로그램은 포트라는 인터페이스를 통해 외부 구성 요소와 통신하고 어댑터를 사용하여 이러한 구성 요소와의 기술 교환을 변환합니다.

목적

육각형 아키텍처 패턴은 데이터베이스 또는 외부 API에 액세스하는 코드와 같은 관련 인프라 코드로부터 비즈니스 로직 (도메인 로직) 을 분리하는 데 사용됩니다. 이 패턴은 외부 서비스와의 통합이 필요한 AWS Lambda 기능에 대해 느슨하게 연결된 비즈니스 로직 및 인프라 코드를 만드는 데 유용합니다. 기존 아키텍처에서는 일반적으로 비즈니스 로직을 데이터베이스 계층에 저장 프로시저 및 사용자 인터페이스로 내장하는 것이 일반적입니다. 이러한 방식은 비즈니스 로직 내에서 UI별 구조를 사용하는 것과 함께 밀접하게 연결된 아키텍처로 이어져 데이터베이스 마이그레이션 및 사용자 경험 (UX) 현대화 노력에 병목 현상을 야기합니다. 육각형 아키텍처 패턴을 사용하면 기술이 아닌 목적에 따라 시스템과 애플리케이션을 설계할 수 있습니다. 이 전략을 통해 데이터베이스, UX 및 서비스 구성 요소와 같은 애플리케이션 구성 요소를 쉽게 교환할 수 있습니다.

적용 가능성

다음과 같은 경우 육각형 아키텍처 패턴을 사용하십시오.

- 애플리케이션 아키텍처를 분리하여 완전히 테스트할 수 있는 구성 요소를 만들고 싶습니다.
- 여러 유형의 클라이언트가 동일한 도메인 로직을 사용할 수 있습니다.
- UI와 데이터베이스 구성 요소에는 애플리케이션 로직에 영향을 주지 않는 주기적인 기술 업데이트가 필요합니다.
- 응용 프로그램에는 여러 입력 공급자와 출력 소비자가 필요하며 응용 프로그램 로직을 사용자 지정하면 코드가 복잡해지고 확장성이 떨어집니다.

문제 및 고려 사항

- 도메인 기반 설계: 육각형 아키텍처는 도메인 기반 설계 (DDD) 와 특히 잘 어울립니다. 각 애플리케이션 구성 요소는 DDD의 하위 도메인을 나타내며, 육각형 아키텍처를 사용하여 애플리케이션 구성 요소 간의 느슨한 결합을 구현할 수 있습니다.
- 테스트 가능성: 육각형 아키텍처는 설계상 입력과 출력에 추상화를 사용합니다. 따라서 고유한 느슨한 결합으로 인해 단위 테스트를 작성하고 분리하여 테스트하는 것이 더 쉬워집니다.
- 복잡성: 비즈니스 로직과 인프라 코드를 분리하는 복잡성을 신중하게 다룰 경우 민첩성, 테스트 범위 및 기술 적응성과 같은 큰 이점을 가져올 수 있습니다. 그렇지 않으면 문제 해결이 복잡해질 수 있습니다.
- 유지 관리 오버헤드: 아키텍처를 플러그 가능하게 만드는 추가 어댑터 코드는 애플리케이션 구성 요소가 쓰기 위해 여러 입력 소스와 출력 대상이 필요한 경우 또는 시간이 지남에 따라 입력 및 출력 데이터 저장소가 변경되어야 하는 경우에만 정당화됩니다. 그렇지 않으면 어댑터가 유지 관리해야 할 또 다른 계층이 되어 유지 관리 오버헤드가 발생합니다.
- 지연 문제: 포트와 어댑터를 사용하면 또 다른 계층이 추가되어 지연이 발생할 수 있습니다.

구현

육각형 아키텍처는 인프라 코드 및 애플리케이션을 UI, 외부 API, 데이터베이스 및 메시지 브로커와 통합하는 코드로부터 애플리케이션 및 비즈니스 로직을 분리할 수 있도록 지원합니다. 포트와 어댑터를 통해 비즈니스 로직 구성 요소를 애플리케이션 아키텍처의 다른 구성 요소 (예: 데이터베이스) 에 쉽게 연결할 수 있습니다.

포트는 기술에 구애받지 않고 애플리케이션 구성 요소로 들어가는 진입점입니다. 이러한 사용자 지정 인터페이스는 인터페이스를 구현하는 사용자 또는 대상에 관계없이 외부 행위자가 애플리케이션 구성 요소와 통신할 수 있도록 하는 인터페이스를 결정합니다. 이는 다양한 유형의 장치가 USB 어댑터를 사용하기만 하면 USB 포트를 통해 컴퓨터와 통신할 수 있는 방식과 비슷합니다.

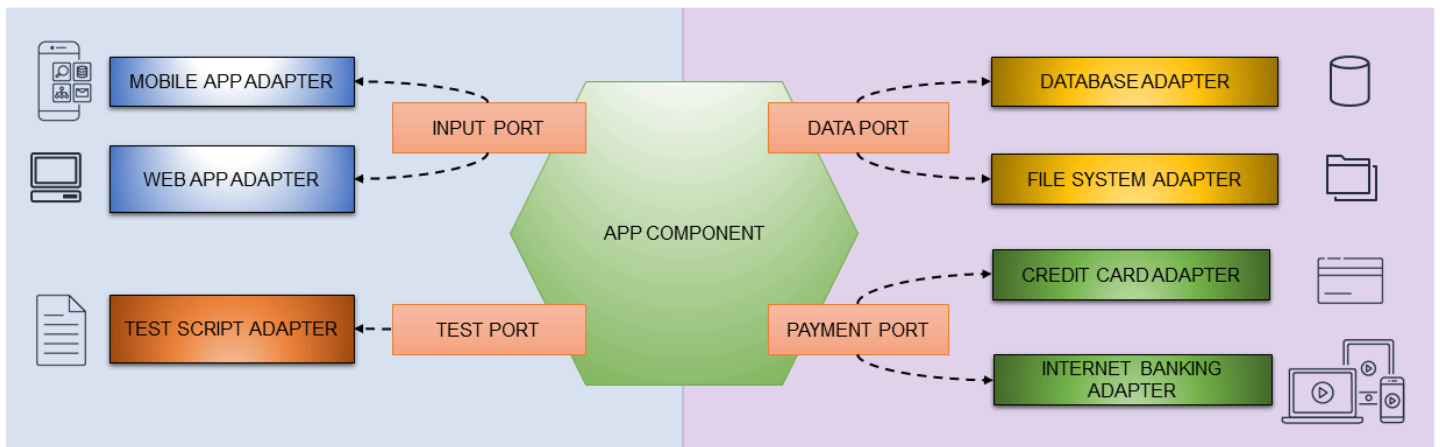
어댑터는 특정 기술을 사용하여 포트를 통해 응용 프로그램과 상호 작용합니다. 어댑터는 이러한 포트에 연결하여 포트에서 데이터를 수신하거나 포트에 데이터를 제공하고 추가 처리를 위해 데이터를 변환합니다. 예를 들어 REST 어댑터를 사용하면 액터가 REST API를 통해 애플리케이션 구성 요소와 통신할 수 있습니다. 포트나 애플리케이션 구성 요소에 대한 위험 없이 포트에 여러 개의 어댑터가 있을 수 있습니다. 이전 예제를 확장하기 위해 동일한 포트에 GraphQL 어댑터를 추가하면 액터가 REST API, 포트 또는 애플리케이션에 영향을 주지 않고 GraphQL API를 통해 애플리케이션과 상호 작용할 수 있는 추가 수단을 제공합니다.

포트는 애플리케이션에 연결되고 어댑터는 외부 세계와의 연결 역할을 합니다. 포트를 사용하여 느슨하게 결합된 애플리케이션 구성 요소를 만들고 어댑터를 변경하여 종속 구성 요소를 교환할 수 있습니다. 따라서 애플리케이션 구성 요소가 상황을 인식하지 않고도 외부 입력 및 출력과 상호 작용할 수 있습니다. 구성 요소는 모든 수준에서 교환할 수 있으므로 자동화된 테스트가 용이합니다. 테스트를 수행하기 위해 전체 환경을 프로비저닝하는 대신 인프라 코드에 대한 종속성 없이 구성 요소를 독립적으로 테스트할 수 있습니다. 애플리케이션 로직은 외부 요인에 의존하지 않으므로 테스트가 간소화되고 종속성을 모의하기가 더 쉬워집니다.

예를 들어, 느슨하게 결합된 아키텍처에서는 응용 프로그램 구성 요소가 데이터 저장소의 세부 정보를 몰라도 데이터를 읽고 쓸 수 있어야 합니다. 애플리케이션 구성 요소의 책임은 인터페이스 (포트) 에 데이터를 공급하는 것입니다. 어댑터는 애플리케이션의 필요에 따라 데이터베이스, 파일 시스템 또는 Amazon S3와 같은 객체 스토리지 시스템이 될 수 있는 데이터 스토어에 쓰는 로직을 정의합니다.

상위 수준 아키텍처

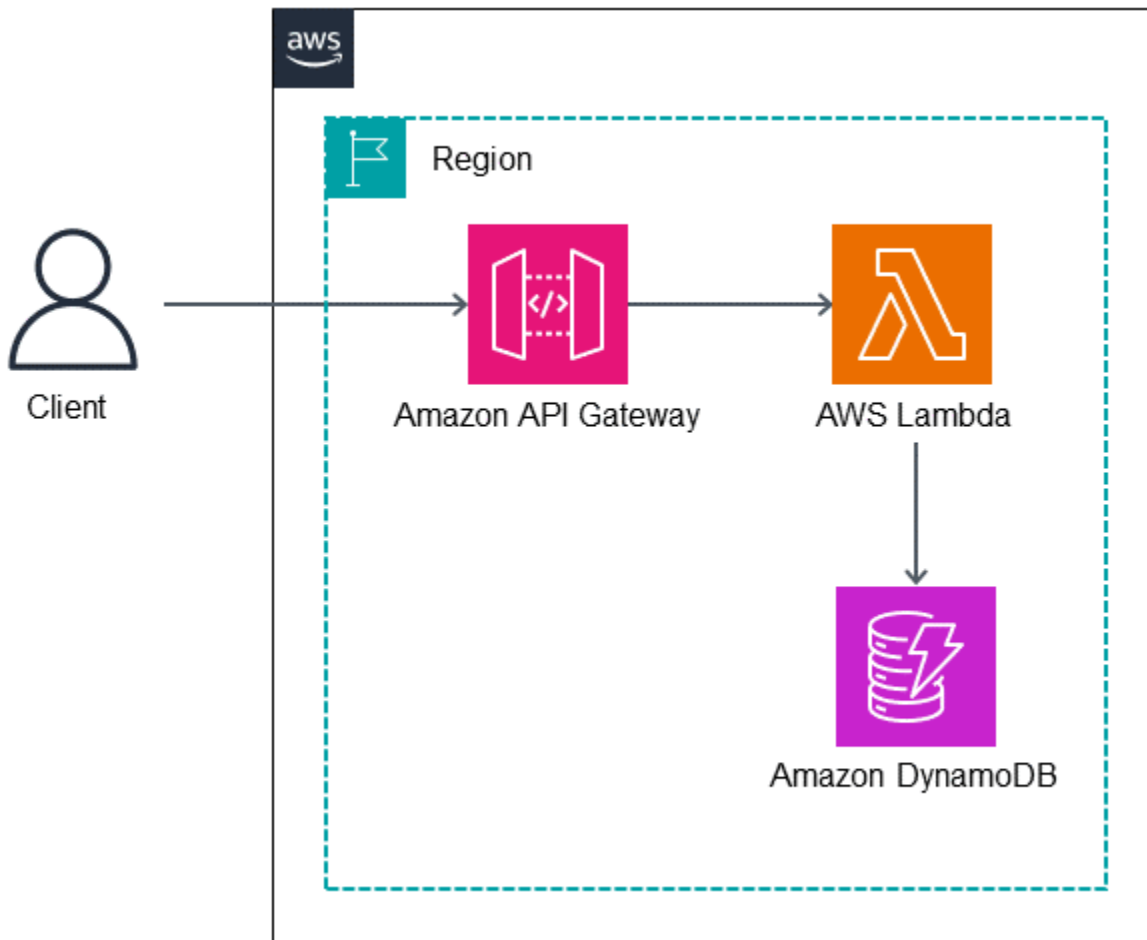
애플리케이션 또는 애플리케이션 구성 요소에는 핵심 비즈니스 로직이 포함되어 있습니다. 다음 다이어그램과 같이 포트에서 명령이나 쿼리를 수신하고 포트를 통해 외부 행위자에게 요청을 보냅니다. 외부 행위자는 어댑터를 통해 구현됩니다.



를 사용한 구현 AWS 서비스

AWS Lambda 함수에는 목표를 달성하기 위해 긴밀하게 연결된 비즈니스 로직과 데이터베이스 통합 코드가 모두 포함되는 경우가 많습니다. 육각형 아키텍처 패턴을 사용하여 비즈니스 로직과 인프라 코드를 분리할 수 있습니다. 이러한 분리를 통해 데이터베이스 코드에 대한 종속성 없이 비즈니스 로직을 단위 테스트할 수 있으며 개발 프로세스의 민첩성이 향상됩니다.

다음 아키텍처에서 Lambda 함수는 육각형 아키텍처 패턴을 구현합니다. Lambda 함수는 아마존 API 게이트웨이 REST API에 의해 시작됩니다. 이 함수는 비즈니스 로직을 구현하고 DynamoDB 테이블에 데이터를 씁니다.



샘플 코드

이 섹션의 샘플 코드는 Lambda를 사용하여 도메인 모델을 구현하고, 이를 인프라 코드 (예: DynamoDB 액세스 코드) 와 분리하고, 함수에 대한 단위 테스트를 구현하는 방법을 보여줍니다.

도메인 모델

도메인 모델 클래스는 외부 구성 요소나 종속성에 대한 지식이 없으며 비즈니스 로직만 구현합니다. 다음 예제에서 Recipient 클래스는 예약 날짜의 중복을 확인하는 도메인 모델 클래스입니다.

```
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:
    .....

```

입력 포트

RecipientInputPort 클래스는 수신자 클래스에 연결하고 도메인 로직을 실행합니다.

```

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 어댑터 클래스

DDBRecipientAdapter 클래스는 DynamoDB 테이블에 대한 액세스를 구현합니다.

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

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

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

Lambda get_recipient_input_port 함수는 클래스의 인스턴스를 위한 팩토리입니다.

RecipientInputPort 관련 어댑터 인스턴스를 사용하여 출력 포트 클래스의 인스턴스를 생성합니다.

```
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
    }),
}
```

유닛 테스트

모의 클래스를 삽입하여 도메인 모델 클래스의 비즈니스 로직을 테스트할 수 있습니다. 다음 예제는 도메인 모델 Recipient 클래스에 대한 단위 테스트를 제공합니다.

```
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 리포지토리

이 패턴의 샘플 아키텍처를 완전히 구현하려면 <https://github.com/aws-samples/aws-lambda-domain-model-sample>의 GitHub 리포지토리를 참조하십시오.

관련 콘텐츠

- [육각형 아키텍처, 알리스테어](#) [콕번의 기사](#)
- [를 통한 진화 아키텍처 개발](#) (일본어 블로그 게시물) [AWS LambdaAWS](#)

비디오

다음 동영상 (일본어)에서는 Lambda 함수를 사용하여 도메인 모델을 구현할 때 육각형 아키텍처를 사용하는 방법을 설명합니다.

게시-구독 패턴

Intent

게시-구독 패턴은 pub-sub 패턴이라고도 하며, 메시지 발신자(게시자)와 관심 있는 수신자(구독자)를 서로 분리하는 메시징 패턴입니다. 이 패턴은 메시지 브로커 또는 라우터라는 중개자(메시지 인프라)를 통해 메시지나 이벤트를 게시하여 비동기 통신을 구현합니다. 게시-구독 패턴은 메시지 전송을 메시지 인프라가 처리하게 하여, 발신자가 핵심 메시지 처리에 집중할 수 있도록 함으로써 발신자의 확장성과 응답 성능을 높입니다.

목적

분산 아키텍처에서 시스템 구성 요소는 시스템 내에서 이벤트가 발생할 때 다른 구성 요소에 정보를 제공해야 하는 경우가 많습니다. 게시-구독 패턴은 문제를 구분하여 애플리케이션이 핵심 기능에 집중할 수 있도록 합니다. 따라서 메시지 인프라가 메시지 라우팅 및 안정적인 전송과 같은 통신을 처리하는 동안 애플리케이션이 핵심 기능에 집중할 수 있습니다. 게시-구독 패턴을 사용하면 비동기 메시징에서 게시자와 구독자를 분리할 수 있습니다. 구독자가 모르는 사이에 게시자가 메시지를 보낼 수도 있습니다.

적용 가능성

다음과 같은 경우 게시-구독 패턴을 사용합니다.

- 단일 메시지에 여러 워크플로가 사용되어 병렬 처리가 필요합니다.
- 메시지를 여러 구독자에게 브로드캐스트하거나 수신자의 실시간 응답을 받을 필요가 없습니다.
- 시스템 또는 애플리케이션이 데이터 또는 상태의 최종 일관성을 허용할 수 있습니다.
- 애플리케이션 또는 구성 요소가 다른 언어, 프로토콜 또는 플랫폼을 사용하는 애플리케이션 또는 서비스와 통신해야 합니다.

문제 및 고려 사항

- 구독자 가용성: 게시자는 구독자가 수신 중인지 여부를 알지 못하며 구독자가 수신하고 있지 않을 수도 있습니다. 게시된 메시지는 일시적이며 구독자가 수신할 수 없는 경우 삭제될 수 있습니다.

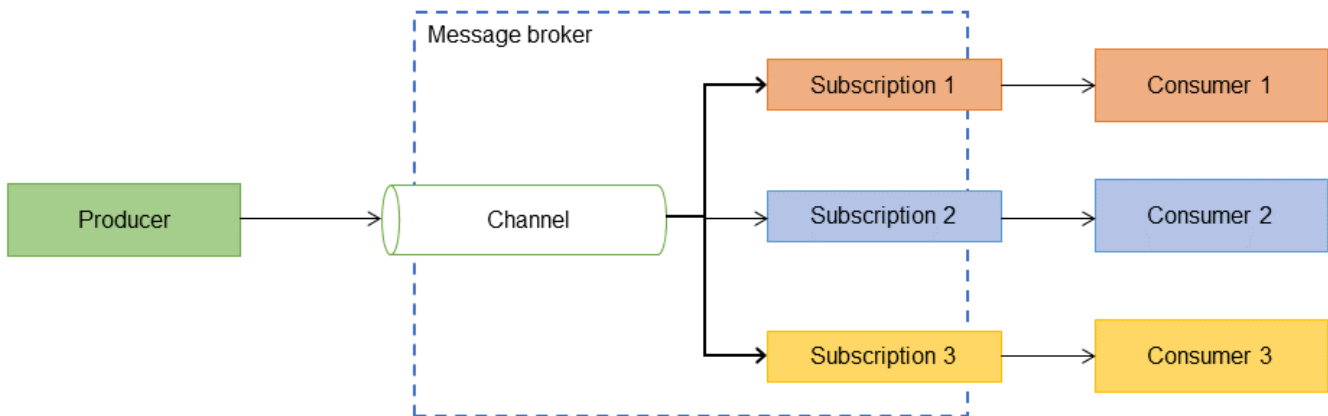
- 메시지 전송 보장: 일반적으로 게시-구독 패턴으로는 모든 구독자 유형에 대한 메시지 전송을 보장할 수 없지만, Amazon Simple Notification Service(SNS)와 같은 특정 서비스는 일부 구독자 하위 집합에 대해 정확히 한 번의 전송을 제공할 수 있습니다.
- Time to live(TTL): 메시지는 수명이 있으며 일정 기간 내에 처리되지 않으면 만료됩니다. 따라서 게시된 메시지를 대기열에 추가하여 메시지가 계속 유지되도록 하고 TTL 기간 이후에도 처리되도록 하는 것이 좋습니다.
- 메시지 연관성: 생산자는 메시지 데이터의 일부로 연관성 기간을 설정할 수 있으며, 이 날짜 이후에는 메시지를 삭제할 수 있습니다. 메시지 처리 방법을 결정하기 전에 소비자가 이 정보를 검토하도록 설계하는 것이 좋습니다.
- 최종 일관성: 메시지가 게시되는 시간과 구독자가 메시지를 소비하는 시간 사이에는 지연이 발생합니다. 그 덕분에 강력한 일관성이 요구될 때 구독자 데이터 스토어가 일관성을 유지할 수 있게 됩니다. 생산자와 소비자가 거의 실시간으로 상호 작용해야 하는 경우에도 최종 일관성이 문제가 될 수 있습니다.
- 단방향 통신: 게시-구독 패턴은 단방향으로 간주됩니다. 반환 구독 채널을 통한 양방향 메시징이 필요한 애플리케이션에서 동기 응답이 필요한 경우 요청-응답 패턴을 사용하는 것이 좋습니다.
- 메시지 순서: 메시지 순서는 보장되지 않습니다. 소비자가 순서에 따른 메시지를 요구하는 경우 Amazon SNS FIFO 주제를 사용하여 순서를 보장하는 것이 좋습니다.
- 메시지 중복: 메시징 인프라에 따라 중복 메시지가 소비자에게 전송될 수 있습니다. 따라서 소비자는 중복 메시지를 처리할 수 있는 능력을 갖추도록 설계되어야 합니다. 또는 Amazon SNS FIFO 주제를 사용하여 정확히 한 번의 전송을 보장할 수도 있습니다.
- 메시지 필터링: 소비자는 생산자가 게시한 메시지 중 일부에만 관심을 보이는 경우가 많습니다. 주제 또는 콘텐츠 필터를 제공하여 구독자가 수신하는 메시지를 필터링하거나 범위를 좁힐 수 있는 메커니즘을 제공합니다.
- 메시지 재생: 메시지 재생 기능은 메시징 인프라에 따라 달라질 수 있습니다. 사용 사례에 따라 맞춤형 구현을 제공할 수도 있습니다.
- DLQ(Dead Letter Queue): 우편 시스템에서 배달 불능 우편물과는 배달이 불가능한 우편물을 처리하는 부서입니다. pub/sub 메시징에서 DLQ(Dead Letter Queue)는 구독한 엔드포인트로 배달할 수 없는 메시지를 저장하는 대기열입니다.

구현

전반적인 아키텍처

게시-구독 패턴에서는 메시지 브로커 또는 라우터라고 하는 비동기 메시징 하위 시스템이 구독을 추적합니다. 생산자가 이벤트를 게시하면 메시징 인프라가 각 소비자에게 메시지를 보냅니다. 메시지는 구독자에게 전송된 후 메시지 인프라에서 제거되므로 다시 재생할 수 없으며 새 구독자는 해당 이벤트를 볼 수 없습니다. 메시지 브로커 또는 라우터는 다음과 같은 방법으로 이벤트 생산자를 메시지 소비자와 분리합니다.

- 생산자가 정의된 메시지 형식을 사용하여, 메시지로 패키징된 이벤트를 게시할 수 있는 입력 채널을 제공합니다.
- 구독별 출력 채널 생성. 구독은 소비자가 특정 입력 채널과 관련된 이벤트 메시지를 수신하는 연결입니다.
- 이벤트가 게시될 때 모든 소비자를 위해 입력 채널에서 출력 채널로 메시지를 복사합니다.



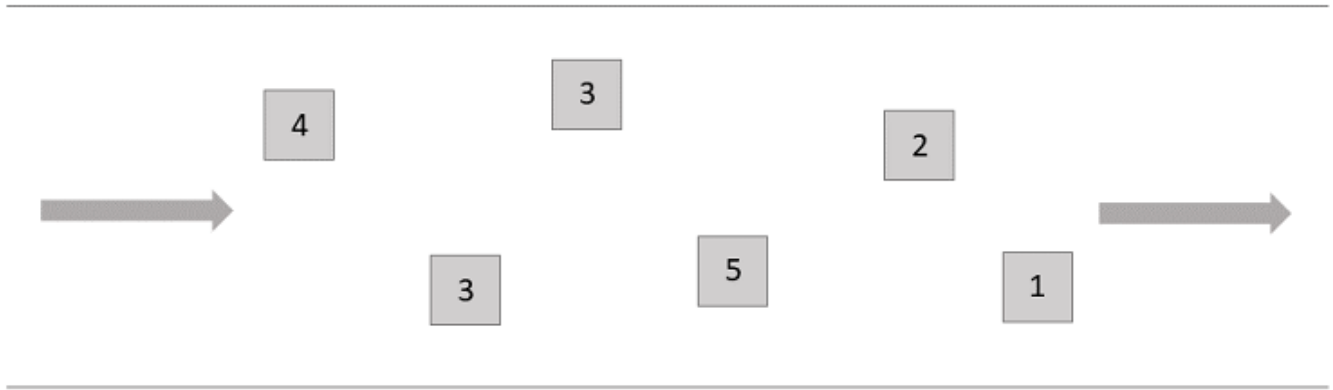
AWS 서비스를 사용한 구현

Amazon SNS

Amazon SNS는 완전관리형 게시자-구독자 서비스로, 분산된 애플리케이션을 분리하기 위한 애플리케이션 간(A2A) 메시징 기능을 제공합니다. 또한 SMS, 이메일 및 기타 푸시 알림 전송을 위한 애플리케이션-사람 간(A2P) 메시징 기능도 제공합니다.

Amazon SNS는 표준 주제와 선입선출(FIFO)이라는 두 가지 유형의 주제를 제공합니다.

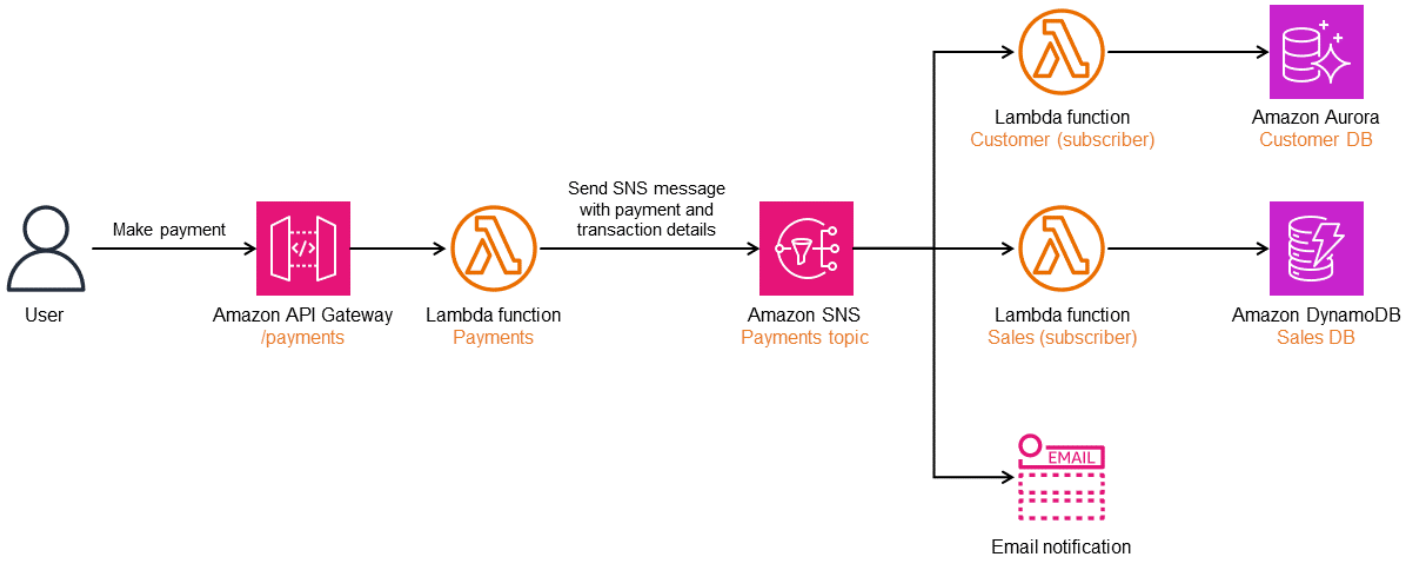
- 표준 주제는 초당 메시지 수에 제한이 없으며 최선의 순서 지정 및 중복 제거 기능을 제공합니다.



- FIFO 주제는 엄격한 순서 지정 및 중복 제거 기능을 제공하며 FIFO 주제당 초당 최대 300건의 메시지 또는 초당 10MB(둘 중 먼저 도래하는 기준)를 지원합니다.

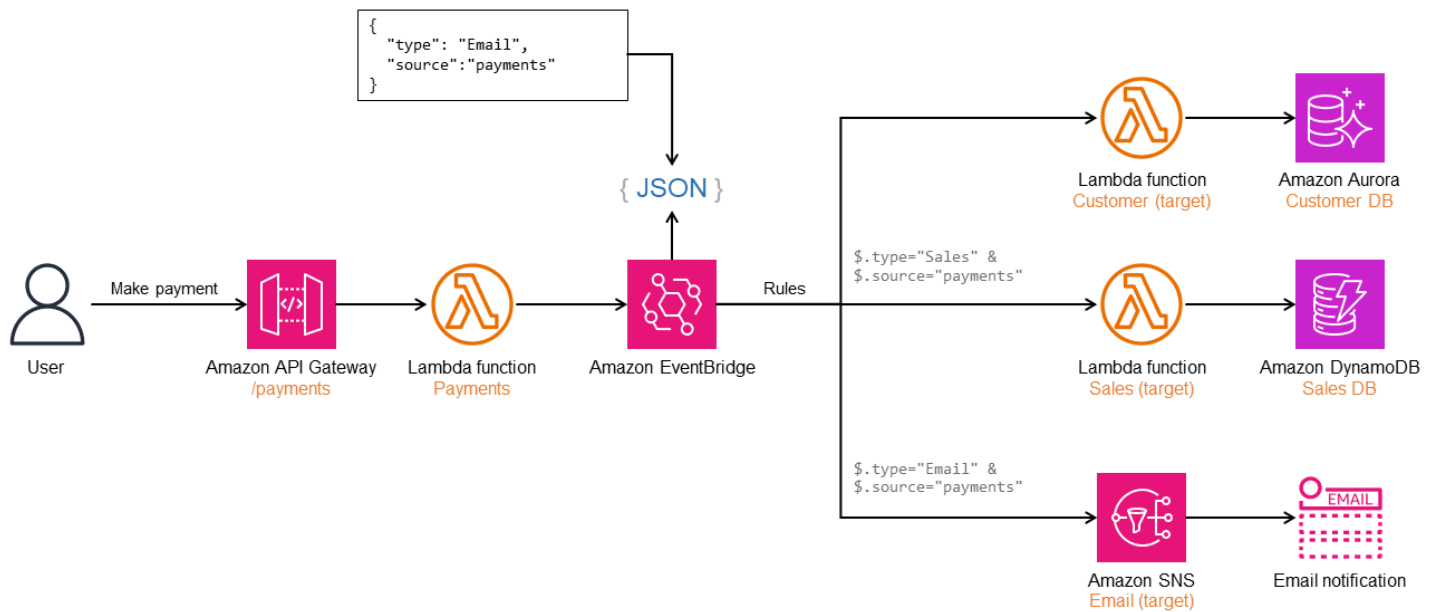


다음 그림은 Amazon SNS를 사용하여 게시-구독 패턴을 구현하는 방법을 보여줍니다. 사용자가 결제를 완료하면 Payments Lambda 함수를 통해 Payments SNS 주제로 SNS 메시지가 전송됩니다. 이 SNS 주제의 구독자는 셋입니다. 각 구독자는 메시지 사본을 수신하여 처리합니다.



Amazon EventBridge

여러 생산자의 메시지를 다양한 프로토콜을 통해 구독한 소비자에게 라우팅하거나, 직접 및 팬아웃 구독을 사용하는 등 복잡한 라우팅이 필요한 경우 Amazon EventBridge를 사용할 수 있습니다. EventBridge는 콘텐츠 기반 라우팅, 필터링, 시퀀싱, 분할 또는 집계도 지원합니다. 다음 그림에서 EventBridge는 이벤트 규칙을 사용하여 구독자를 정의하는 게시-구독 패턴 버전을 구축하는 데 사용됩니다. 사용자가 결제를 완료하면 Payments Lambda 함수는 서로 다른 대상을 가리키는 세 가지 규칙이 있는 사용자 지정 스키마를 기반으로 한 기본 이벤트 버스를 사용하여 EventBridge에 메시지를 보냅니다. 각 마이크로서비스는 메시지를 처리하고 필요한 작업을 수행합니다.



워크숍

- [Building event-driven architectures on AWS](#)
- [Send Fanout Event Notifications with Amazon Simple Queue Service \(Amazon SQS\) and Amazon Simple Notification Service \(Amazon SNS\)](#)

블로그 참조

- [Choosing between messaging services for serverless applications](#)
- [Designing durable serverless apps with DLQs for Amazon SNS, Amazon SQS, AWS Lambda](#)
- [Simplify your pub/sub messaging with Amazon SNS message filtering](#)

관련 콘텐츠

- [Features of pub/sub messaging](#)

백오프 패턴으로 재시도

Intent

백오프 패턴을 사용한 재시도는 일시적인 오류로 인해 실패한 작업을 투명하게 재시도하여 애플리케이션 안정성을 향상시킵니다.

동기부여

분산 아키텍처에서는 서비스 조절, 일시적인 네트워크 연결 손실 또는 일시적인 서비스 가용성 장애로 인해 일시적인 오류가 발생할 수 있습니다. 이러한 일시적인 오류로 인해 실패한 작업을 자동으로 재시도하면 사용자 경험과 애플리케이션 레질리언스가 향상됩니다. 그러나 자주 재시도하면 네트워크 대역폭에 과부하가 걸리고 경합이 발생할 수 있습니다. 지수 백오프는 지정된 재시도 횟수만큼 대기 시간을 늘려 작업을 재시도하는 기술입니다.

적용 가능성

다음과 같은 경우 백오프 패턴과 함께 재시도를 사용하십시오.

- 서비스는 과부하를 방지하기 위해 요청을 자주 제한하여 다음과 같은 결과를 초래합니다. 429 요청이 너무 많음 호출 프로세스에 대한 예외.
- 네트워크는 분산 아키텍처에서 눈에 띄지 않는 역할을 하며 일시적인 네트워크 문제로 인해 장애가 발생합니다.
- 호출 중인 서비스를 일시적으로 사용할 수 없어 장애가 발생했습니다. 이 패턴을 사용하여 백오프 타임아웃을 유발하지 않는 한 자주 재시도하면 서비스 성능이 저하될 수 있습니다.

문제 및 고려 사항

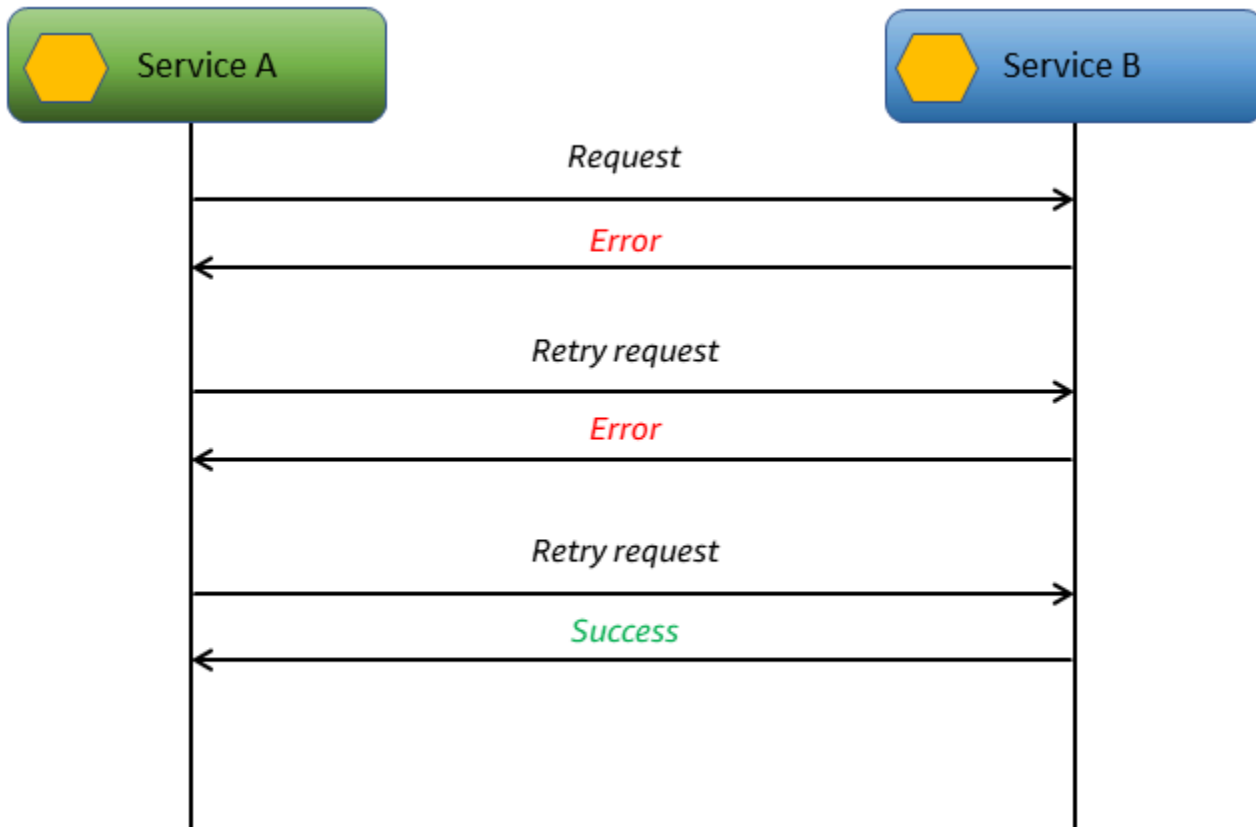
- 임피던스: 메시지를 여러 번 호출해도 시스템 상태를 한 번 호출한 것과 동일한 효과가 있는 경우 해당 작업은 무력한 것으로 간주됩니다. 백오프 패턴과 함께 재시도를 사용할 때는 작업이 동일해야 합니다. 그렇지 않으면 부분 업데이트로 인해 시스템 상태가 손상될 수 있습니다.
- 네트워크 대역폭: 너무 많은 재시도가 네트워크 대역폭을 차지하여 응답 시간이 느려질 경우 서비스 성능이 저하될 수 있습니다.

- 실패가 빠른 시나리오: 일시적이지 않은 오류의 경우 장애 원인을 파악할 수 있다면 회로 차단기 패턴을 사용하여 빠르게 실패하는 것이 더 효율적입니다.
- 백오프 속도: 기하급수적 백오프를 도입하면 서비스 타임아웃에 영향을 주어 최종 사용자의 대기 시간이 길어질 수 있습니다.

구현

고수준 아키텍처

다음 다이어그램은 성공적인 응답이 반환될 때까지 서비스 A가 서비스 B에 대한 호출을 재시도하는 방법을 보여줍니다. 서비스 B가 몇 번의 시도 후에도 성공적인 응답을 반환하지 않으면 서비스 A는 재시도를 중지하고 호출자에게 실패를 반환할 수 있습니다.



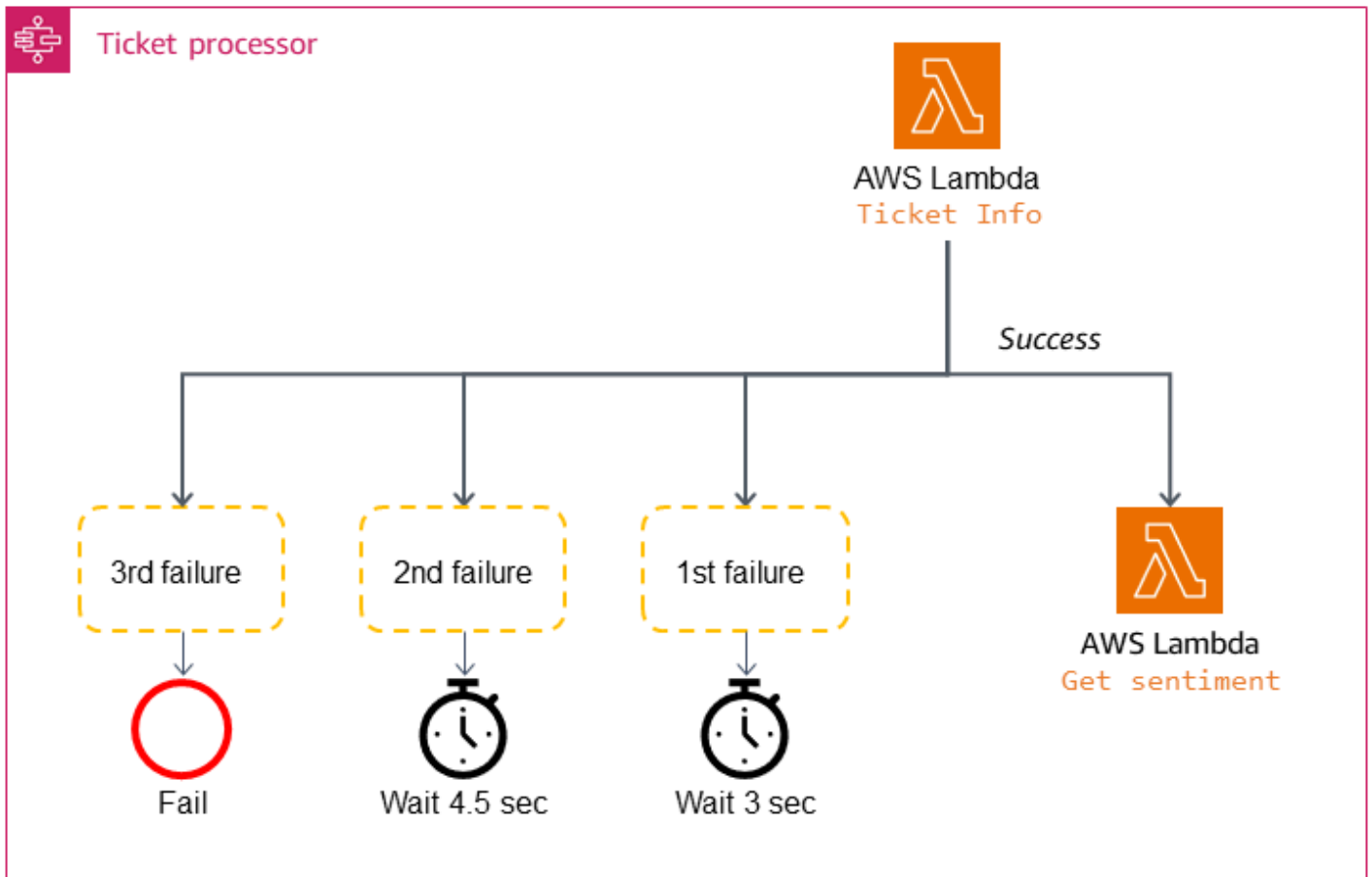
를 사용한 구현AWS서비스

다음 다이어그램은 고객 지원 플랫폼의 티켓 처리 워크플로를 보여줍니다. 불만이 있는 고객의 티켓은 티켓 우선 순위를 자동으로 높여 신속하게 처리됩니다. 더Ticket infoLambda 함수는 티켓 세부 정

보를 추출하고 다음을 호출합니다. Get sentiment 란다 함수 더 Get sentiment Lambda 함수는 설명을 다음 주소로 전달하여 고객 감정을 확인합니다. [아마존 이해](#) (표시되지 않음).

전화를 걸면 Get sentiment Lambda 함수가 실패하면 워크플로가 작업을 세 번 재시도합니다. AWS Step Functions 백오프 값을 구성할 수 있도록 하여 지수적 백오프를 허용합니다.

이 예에서는 최대 세 번의 재시도가 1.5초의 증가 비율로 구성됩니다. 첫 번째 재시도가 3초 후에 발생하는 경우 두 번째 재시도는 $3 \times 1.5\text{초} = 4.5\text{초}$ 후에 발생하고 세 번째 재시도는 $4.5 \times 1.5\text{초} = 6.75\text{초}$ 후에 발생합니다. 세 번째 재시도가 실패하면 워크플로가 실패합니다. 백오프 로직에는 사용자 지정 코드가 필요하지 않습니다. 이 코드는 다음과 같이 구성으로 제공됩니다. AWS Step Functions.



샘플 코드

다음 코드는 백오프 패턴을 사용한 재시도의 구현을 보여줍니다.

```
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 occurred, so stop calling the API
        default:
            retry = false;
            break;
    }
    retries++;
} while (retry && retries < MAX_RETRIES);
}
```

GitHub저장소

이 패턴에 대한 샘플 아키텍처의 전체 구현은 다음을 참조하십시오. GitHub리포지토리 주소 <https://github.com/aws-samples/retry-with-backoff>.

관련 콘텐츠

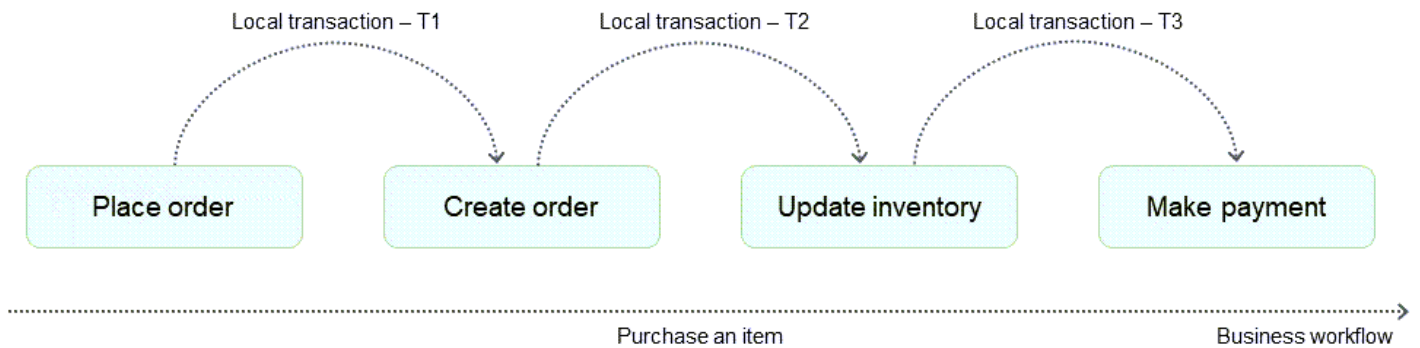
- [타임아웃, 재시도 및 지터로 인한 백오프](#)(아마존 빌더 라이브러리)

Saga 패턴

Saga는 일련의 로컬 트랜잭션으로 구성됩니다. Saga의 각 로컬 트랜잭션은 데이터베이스를 업데이트하고 다음 로컬 트랜잭션을 트리거합니다. 트랜잭션이 실패하면 Saga에서는 보정 트랜잭션을 실행하여 이전 트랜잭션에서 수행한 데이터베이스 변경 내용을 되돌립니다.

이러한 로컬 트랜잭션 시퀀스는 연속성 및 보정 원칙을 사용하여 비즈니스 워크플로를 실현하는 데 도움이 됩니다. 연속성 원칙은 워크플로의 순방향 복구를 결정하는 반면 보정 원칙은 역방향 복구를 결정합니다. 트랜잭션의 어느 단계에서든 업데이트가 실패하면 Saga에서는 연속성(트랜잭션 재시도) 또는 보정(이전 데이터 상태로 되돌리기)을 위한 이벤트를 게시합니다. 따라서 데이터 스토어 전반에서 데이터 무결성과 일관성이 유지됩니다.

예를 들어 사용자가 온라인 소매업체에서 책을 구매하는 경우 프로세스는 주문 생성, 재고 업데이트, 결제, 배송 등 비즈니스 워크플로를 나타내는 일련의 트랜잭션으로 구성됩니다. 이 워크플로를 완료하기 위해 분산 아키텍처는 일련의 로컬 트랜잭션을 실행하여 주문 데이터베이스에서 주문을 생성하고, 재고 데이터베이스를 업데이트하며, 결제 데이터베이스를 업데이트합니다. 프로세스가 성공하면 다음 다이어그램에서 보듯이 이러한 거래가 순차적으로 간접 호출되어 비즈니스 워크플로가 완료됩니다. 하지만 이러한 로컬 트랜잭션 중 하나라도 실패할 경우 시스템은 적절한 다음 단계, 즉 순방향 복구 또는 역방향 복구를 결정해야 합니다.



다음 두 시나리오는 다음 단계로 순방향 복구를 수행할지 역방향 복구를 수행할지를 결정하는 데 도움이 됩니다.

- 기본 인프라에 문제가 발생하여 트랜잭션이 실패하는 플랫폼 수준 오류. 이 경우 Saga 패턴은 로컬 트랜잭션을 재시도하고 비즈니스 프로세스를 계속하여 순방향 복구를 수행할 수 있습니다.
- 잘못된 결제로 인해 결제 서비스가 실패하는 애플리케이션 수준 오류. 이 경우 Saga 패턴은 보정 트랜잭션을 실행하여 재고 및 주문 데이터베이스를 업데이트하고 이전 상태로 복원함으로써 역방향 복구를 수행할 수 있습니다.

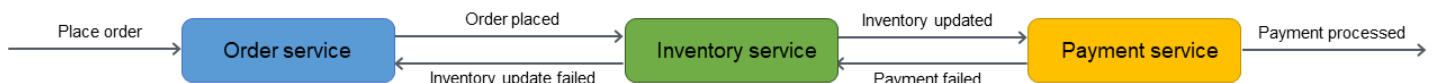
Saga 패턴은 비즈니스 워크플로를 처리하고 순방향 복구를 통해 바람직한 최종 상태에 도달하도록 보장합니다. 장애 발생 시에는 역방향 복구를 사용하여 로컬 트랜잭션을 되돌려 데이터 일관성 문제를 방지합니다.

Saga 패턴에는 코레오그래피와 오케스트레이션이라는 두 가지 변형이 있습니다.

Saga 코레오그래피

Saga 코레오그래피 패턴은 마이크로서비스에서 게시한 이벤트에 따라 달라집니다. Saga 참가자(마이크로서비스)는 이벤트를 구독하고 이벤트 트리거에 따라 동작합니다. 예를 들어 다음 다이어그램의 주문 서비스는 OrderPlaced 이벤트를 발생시킵니다. 인벤토리 서비스는 해당 이벤트를 구독하고 OrderPlaced 이벤트가 발생하면 인벤토리를 업데이트합니다. 마찬가지로 참가자 서비스는 발생한 이벤트의 컨텍스트에 따라 작동합니다.

Saga 코레오그래피 패턴은 Saga의 참가자가 소수에 불과하고 단일 장애 지점이 없는 간단한 구현이 필요한 경우에 적합합니다. 참가자가 더 추가되면 이 패턴을 사용하여 참가자 간의 종속성을 추적하기가 더 어려워집니다.

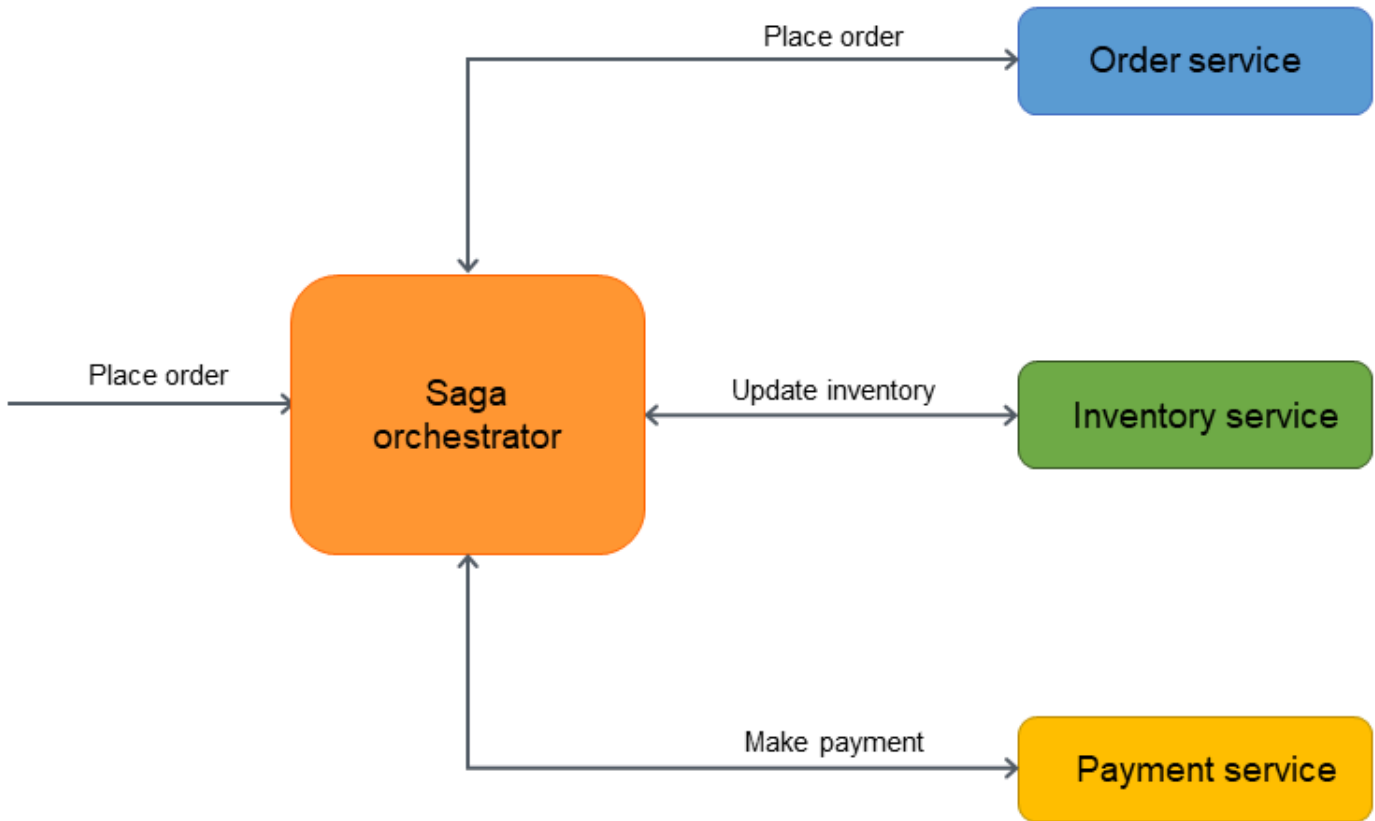


자세한 리뷰는 이 가이드의 [Saga 코레오그래피](#) 섹션을 참조하세요.

Saga 오케스트레이션

Saga 오케스트레이션 패턴에는 오케스트레이터라는 중앙 코디네이터가 있습니다. Saga 오케스트레이터는 전체 트랜잭션 수명 주기를 관리하고 조정합니다. 트랜잭션을 완료하기 위해 수행해야 하는 일련의 단계를 인식합니다. 단계를 실행하기 위해 참가자 마이크로서비스에 메시지를 보내 작업을 수행합니다. 참가자 마이크로서비스는 작업을 완료하고 오케스트레이터로 메시지를 다시 보냅니다. 오케스트레이터는 수신한 메시지에 따라 트랜잭션에서 다음으로 실행할 마이크로서비스를 결정합니다.

Saga 오케스트레이션 패턴은 참가자가 많고 Saga 참가자 간에 느슨한 결합이 필요한 경우에 적합합니다. 오케스트레이터는 참가자를 느슨하게 결합하여 복잡한 로직을 캡슐화합니다. 하지만 오케스트레이터는 전체 워크플로를 제어하는 만큼 단일 장애 지점이 될 수 있습니다.



자세한 리뷰는 이 가이드의 [Saga 오케스트레이션](#) 섹션을 참조하세요.

Saga 코레오그래피 패턴

의도

Saga 코레오그래피 패턴은 이벤트 구독을 사용하여 여러 서비스에 걸친 분산 트랜잭션에서 데이터 무결성을 유지하는 데 도움이 됩니다. 분산 트랜잭션에서는 트랜잭션이 완료되기 전에 여러 서비스를 직접적으로 호출할 수 있습니다. 서비스가 여러 데이터 스토어에 데이터를 저장하는 경우 이러한 데이터 스토어 전반에서 데이터 일관성을 유지하기가 어려울 수 있습니다.

목적

트랜잭션은 여러 단계를 포함할 수 있는 단일 작업 단위로, 모든 단계가 완전히 실행되거나 어떤 단계도 실행되지 않기 때문에 데이터 스토어의 일관된 상태가 유지됩니다. 원자성, 일관성, 격리 및 내구성 (ACID)이라는 용어로 트랜잭션의 속성이 정의됩니다. 관계형 데이터베이스는 데이터 일관성을 유지하기 위해 ACID 트랜잭션을 제공합니다.

트랜잭션의 일관성을 유지하기 위해 관계형 데이터베이스는 2단계 커밋(2PC) 방식을 사용합니다. 이는 준비 단계와 커밋 단계로 구성됩니다.

- 준비 단계에서는 조정 프로세스를 통해 트랜잭션의 참여 프로세스(참가자)에 트랜잭션을 커밋 또는 롤백할 것을 약속하도록 요청합니다.
- 커밋 단계에서 조정 프로세스는 참가자에게 트랜잭션을 커밋하도록 요청합니다. 참가자가 준비 단계에서 커밋에 동의하지 않으면 트랜잭션이 롤백됩니다.

[database-per-service 설계 패턴](#)을 따르는 분산 시스템에서는 2단계 커밋이 선택 사항이 아닙니다. 이는 각 트랜잭션이 다양한 데이터베이스에 분산되어 있고 관계형 데이터 스토어의 2단계 커밋과 유사한 프로세스를 조정할 수 있는 단일 컨트롤러가 없기 때문입니다. 이 경우 해결 방법 중 하나는 Saga 코레오그래피 패턴을 사용하는 것입니다.

적용 가능성

다음과 같은 경우에 Saga 코레오그래피 패턴을 사용합니다.

- 시스템에서 여러 데이터 스토어에 걸친 분산 트랜잭션의 데이터 무결성과 일관성이 요구됩니다.
- 데이터 스토어(예: NoSQL 데이터베이스)는 ACID 트랜잭션을 제공하기 위한 2PC를 제공하지 않으므로, 단일 트랜잭션 내에서 여러 테이블을 업데이트해야 하며 애플리케이션 경계 내에서 2PC를 구현하기가 복잡합니다.
- 참가자 트랜잭션을 관리하는 중앙 제어 프로세스가 단일 장애 지점이 될 수 있습니다.
- Saga 참가자는 독립적인 서비스이므로 느슨하게 결합되어야 합니다.
- 비즈니스 도메인의 제한된 컨텍스트 간에 커뮤니케이션이 이루어집니다.

문제 및 고려 사항

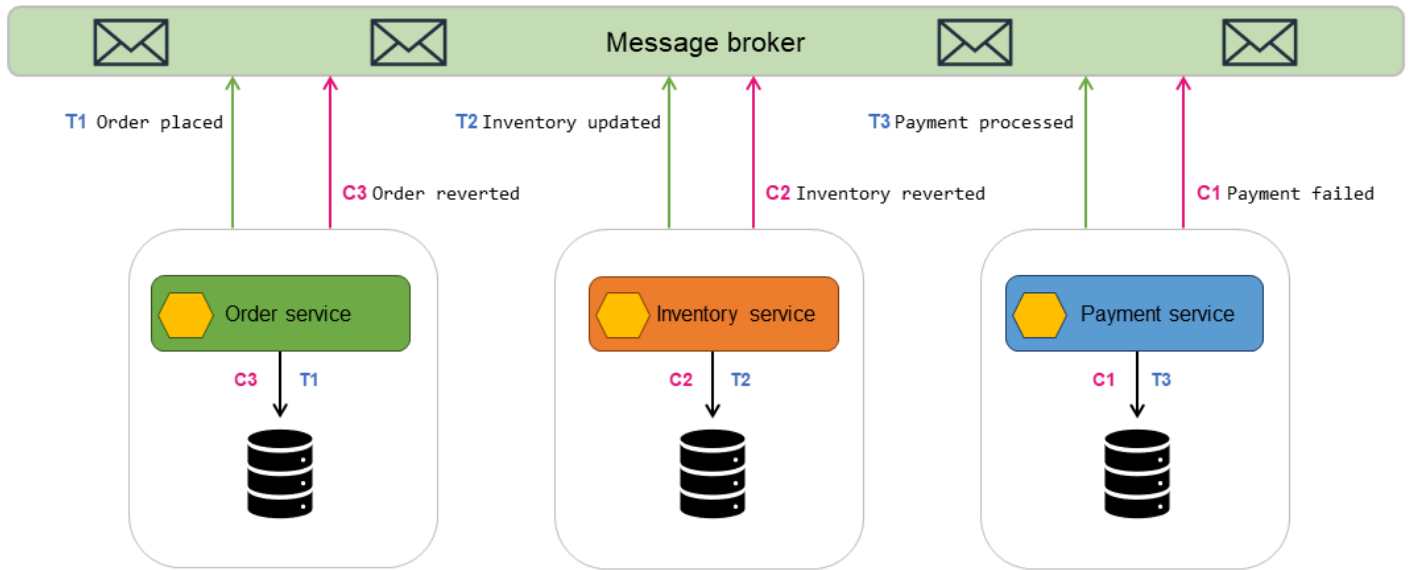
- 복잡성: 마이크로서비스의 수가 증가하면 마이크로서비스 간의 상호 작용이 많아져 Saga 코레오그래피를 관리하기가 어려워질 수 있습니다. 또한 보정 트랜잭션 및 재시도로 인해 애플리케이션 코드의 복잡성이 가중되어 유지 관리 오버헤드를 초래할 수 있습니다. 코레오그래피는 Saga의 참가자가 소수에 불과하고 단일 장애 지점이 없는 간단한 구현이 필요한 경우에 적합합니다. 참가자가 더 추가되면 이 패턴을 사용하여 참가자 간의 종속성을 추적하기가 더 어려워집니다.
- 탄력적인 구현: Saga 코레오그래피에서는 시간 초과, 재시도 및 기타 복원력 패턴을 전반적으로 구현하기가 Saga 오케스트레이션에 비해 더 어렵습니다. 코레오그래피는 오케스트레이터 수준이 아닌 개별 구성 요소에 구현되어야 합니다.

- 주기적 종속성: 참가자들은 서로가 게시한 메시지를 소비합니다. 이로 인해 주기적 종속성이 발생하여 코드 복잡성과 유지 관리 오버헤드가 발생하고 교착 상태가 발생할 수 있습니다.
- 이중 쓰기 문제: 마이크로서비스는 데이터베이스를 자동으로 업데이트하고 이벤트를 게시해야 합니다. 두 작업 중 하나가 실패하면 상태가 일관되지 않을 수 있습니다. 이 문제를 해결하는 방법 중 하나는 [트랜잭션 아웃박스 패턴](#)을 사용하는 것입니다.
- 이벤트 보존: Saga 참가자들은 게시된 이벤트에 따라 작동합니다. 감사, 디버깅 및 재생을 위해 이벤트를 발생한 순서대로 저장하는 것이 중요합니다. 데이터 일관성을 복원하기 위해 시스템 상태를 재생해야 하는 경우 [이벤트 소싱 패턴](#)을 사용하여 이벤트 스토어에 이벤트를 유지할 수 있습니다. 이벤트 스토어는 시스템의 모든 변경 사항을 반영하므로 감사 및 문제 해결 목적으로도 사용할 수 있습니다.
- 최종 일관성: 로컬 트랜잭션을 순차적으로 처리하면 최종 일관성이 유지되며, 이는 강력한 일관성이 필요한 시스템에서는 까다로운 과제가 될 수 있습니다. 정합성 모델에 대한 비즈니스 팀의 기대치를 설정하거나 사용 사례를 재평가하여 강력한 일관성을 제공하는 데이터베이스로 전환함으로써 이 문제를 해결할 수 있습니다.
- 멱등성: Saga 참가자는 예상치 못한 충돌 및 오케스트레이터 장애로 인한 일시적 장애 발생 시, 반복 실행을 허용할 수 있는 멱등성을 갖추어야 합니다.
- 트랜잭션 격리: Saga 패턴은 ACID 트랜잭션의 네 가지 속성 중 하나인 트랜잭션 격리를 지원하지 않습니다. 트랜잭션의 [격리 정도](#)에 따라 다른 동시 트랜잭션이 기반이 되는 데이터에 어느 정도 영향을 미칠 수 있는지 결정됩니다. 트랜잭션을 동시에 오케스트레이션하면 데이터가 부실해질 수 있습니다. 이러한 시나리오를 처리하려면 시맨틱 잠금을 사용하는 것이 좋습니다.
- 관찰성: 관찰성이란 구현 및 오케스트레이션 프로세스의 문제를 해결하기 위한 상세한 로깅과 추적을 의미합니다. 이는 Saga 참가자 수가 증가하여 디버깅이 복잡해질 경우에 중요해집니다. 사가 오케스트레이션에 비해 사가 안무에서는 E nd-to-end 모니터링 및 보고가 더 어렵습니다.
- 지연 문제: Saga가 여러 단계로 구성된 경우 보정 트랜잭션으로 인해 전체 응답 시간에서 지연 시간이 늘어날 수 있습니다. 트랜잭션이 동기식으로 직접 호출을 수행하는 경우 지연 시간이 더 늘어날 수 있습니다.

구현

전반적인 아키텍처

다음 아키텍처 다이어그램에서 Saga 코레오그래피에는 주문 서비스, 인벤토리 서비스, 결제 서비스라는 세 가지 참가자가 참여합니다. 트랜잭션을 완료하려면 T1, T2, T3의 세 단계가 필요합니다. 세 가지 보정 트랜잭션은 데이터를 초기 상태(C1, C2, C3)로 복원합니다.



- 주문 서비스는 데이터베이스를 자동으로 업데이트하고 메시지 브로커에 Order placed 메시지를 게시하는 로컬 트랜잭션 T1을 실행합니다.
- 재고 서비스는 주문 서비스 메시지를 구독하고 주문이 생성되었다는 메시지를 받습니다.
- 재고 서비스는 데이터베이스를 자동으로 업데이트하고 메시지 브로커에 Inventory updated 메시지를 게시하는 로컬 트랜잭션 T2를 실행합니다.
- 결제 서비스는 재고 서비스의 메시지를 구독하고 재고가 업데이트되었다는 메시지를 수신합니다.
- 결제 서비스는 데이터베이스에서 결제 세부 정보를 자동으로 업데이트하고 메시지 브로커에 Payment processed 메시지를 게시하는 로컬 트랜잭션 T3를 실행합니다.
- 결제가 실패하면 결제 서비스는 보정 트랜잭션인 C1을 실행합니다. 이 트랜잭션은 데이터베이스의 결제를 원자적으로 되돌리고 메시지 브로커에 Payment failed 메시지를 게시합니다.
- 보상 트랜잭션 C2와 C3은 데이터 일관성을 복원하기 위해 실행됩니다.

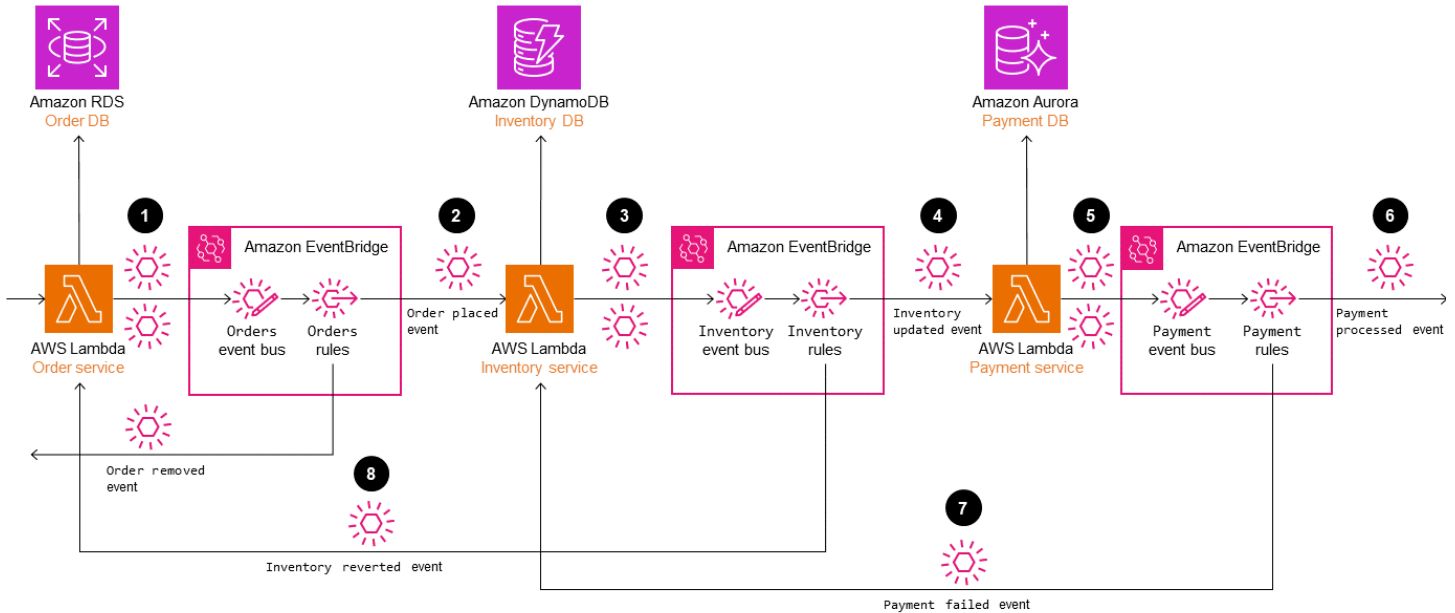
AWS 서비스를 사용한 구현

Amazon을 사용하여 사가 안무 패턴을 구현할 수 있습니다. EventBridge 이벤트를 사용하여 애플리케이션 구성 요소를 연결합니다. 그리고 이벤트 버스 또는 파이프를 통해 이벤트를 처리합니다. 이벤트 버스는 [이벤트](#)를 수신하여 0개 이상의 목적지 또는 대상에 전달하는 라우터입니다. [이벤트 버스와 연결된 규칙](#)은 이벤트가 도착할 때 이벤트를 평가하여 처리를 위해 [대상](#)에 보냅니다.

다음 아키텍처에서:

- 주문 서비스, 재고 서비스, 결제 서비스 등의 마이크로서비스는 Lambda 함수로 구현됩니다.

- 커스텀 EventBridge 버스에 이벤트 버스, Orders 이벤트 버스, Inventory 이벤트 버스의 세 가지가 있습니다. Payment
- Orders 규칙, Inventory 규칙 및 Payment 규칙은 해당 이벤트 버스로 전송된 이벤트를 매칭하고 Lambda 함수를 간접적으로 호출합니다.



성공적인 시나리오에서 주문이 접수되면:

1. 주문 서비스가 요청을 처리하고 이벤트를 Orders 이벤트 버스로 보냅니다.
2. Orders 규칙이 이벤트를 매칭하고 재고 서비스를 시작합니다.
3. 재고 서비스가 인벤토리를 업데이트하고 이벤트를 Inventory 이벤트 버스로 보냅니다.
4. Inventory 규칙이 이벤트를 매칭하고 결제 서비스를 시작합니다.
5. 결제 서비스가 결제를 처리하고 이벤트를 Payment 이벤트 버스로 보냅니다.
6. Payment 규칙이 이벤트와 매칭하고 리스너로 Payment processed 이벤트 알림을 보냅니다.

또는 주문 처리에 문제가 발생할 경우 EventBridge 규칙에 따라 보상 트랜잭션을 시작하여 데이터 업데이트를 되돌려 데이터 일관성과 무결성을 유지할 수 있습니다.

7. 결제가 실패하면 Payment 규칙이 이벤트를 처리하고 재고 서비스를 시작합니다. 재고 서비스가 보상 트랜잭션을 실행하여 재고를 되돌립니다.

8. 재고가 되돌려지면 재고 서비스가 Inventory reverted 이벤트를 Inventory 이벤트 버스로 보냅니다. 이 이벤트는 Inventory 규칙에 따라 처리됩니다. 주문 서비스가 시작되며, 이 서비스는 보정 트랜잭션을 실행하여 주문을 제거합니다.

관련 콘텐츠

- [Saga 오케스트레이션 패턴](#)
- [트랜잭션 아웃박스 패턴](#)
- [백오프 패턴으로 재시도](#)

Saga 오케스트레이션 패턴

의도

Saga 오케스트레이션 패턴은 중앙 코디네이터(오케스트레이터)를 사용하여 여러 서비스에 걸친 분산 트랜잭션에서 데이터 무결성을 유지하는 데 도움을 줍니다. 분산 트랜잭션에서는 트랜잭션이 완료되기 전에 여러 서비스를 직접적으로 호출할 수 있습니다. 서비스가 여러 데이터 스토어에 데이터를 저장하는 경우 이러한 데이터 스토어 전반에서 데이터 일관성을 유지하기가 어려울 수 있습니다.

목적

트랜잭션은 여러 단계를 포함할 수 있는 단일 작업 단위로, 모든 단계가 완전히 실행되거나 어떤 단계도 실행되지 않기 때문에 데이터 스토어의 일관된 상태가 유지됩니다. 원자성, 일관성, 격리 및 내구성(ACID)이라는 용어로 트랜잭션의 속성이 정의됩니다. 관계형 데이터베이스는 데이터 일관성을 유지하기 위해 ACID 트랜잭션을 제공합니다.

트랜잭션의 일관성을 유지하기 위해 관계형 데이터베이스는 2단계 커밋(2PC) 방식을 사용합니다. 이는 준비 단계와 커밋 단계로 구성됩니다.

- 준비 단계에서는 조정 프로세스를 통해 트랜잭션의 참여 프로세스(참가자)에 트랜잭션을 커밋 또는 롤백할 것을 약속하도록 요청합니다.
- 커밋 단계에서 조정 프로세스는 참가자에게 트랜잭션을 커밋하도록 요청합니다. 참가자가 준비 단계에서 커밋에 동의하지 않으면 트랜잭션이 롤백됩니다.

[database-per-service 설계 패턴](#)을 따르는 분산 시스템에서는 2단계 커밋이 선택 사항이 아닙니다. 이는 각 트랜잭션이 다양한 데이터베이스에 분산되어 있고 관계형 데이터 스토어의 2단계 커밋과 유사한

프로세스를 조정할 수 있는 단일 컨트롤러가 없기 때문입니다. 이 경우 해결 방법 중 하나는 Saga 오케스트레이션 패턴을 사용하는 것입니다.

적용 가능성

다음과 같은 경우 Saga 오케스트레이션 패턴을 사용합니다.

- 시스템에서 여러 데이터 스토어에 걸친 분산 트랜잭션의 데이터 무결성과 일관성이 요구됩니다.
- 데이터 스토어가 ACID 트랜잭션을 제공하기 위한 2PC를 제공하지 않으며, 애플리케이션 경계 내에서 2PC를 구현하기가 복잡합니다.
- ACID 트랜잭션을 제공하지 않는 NoSQL 데이터베이스가 있고 단일 트랜잭션 내에서 여러 테이블을 업데이트해야 합니다.

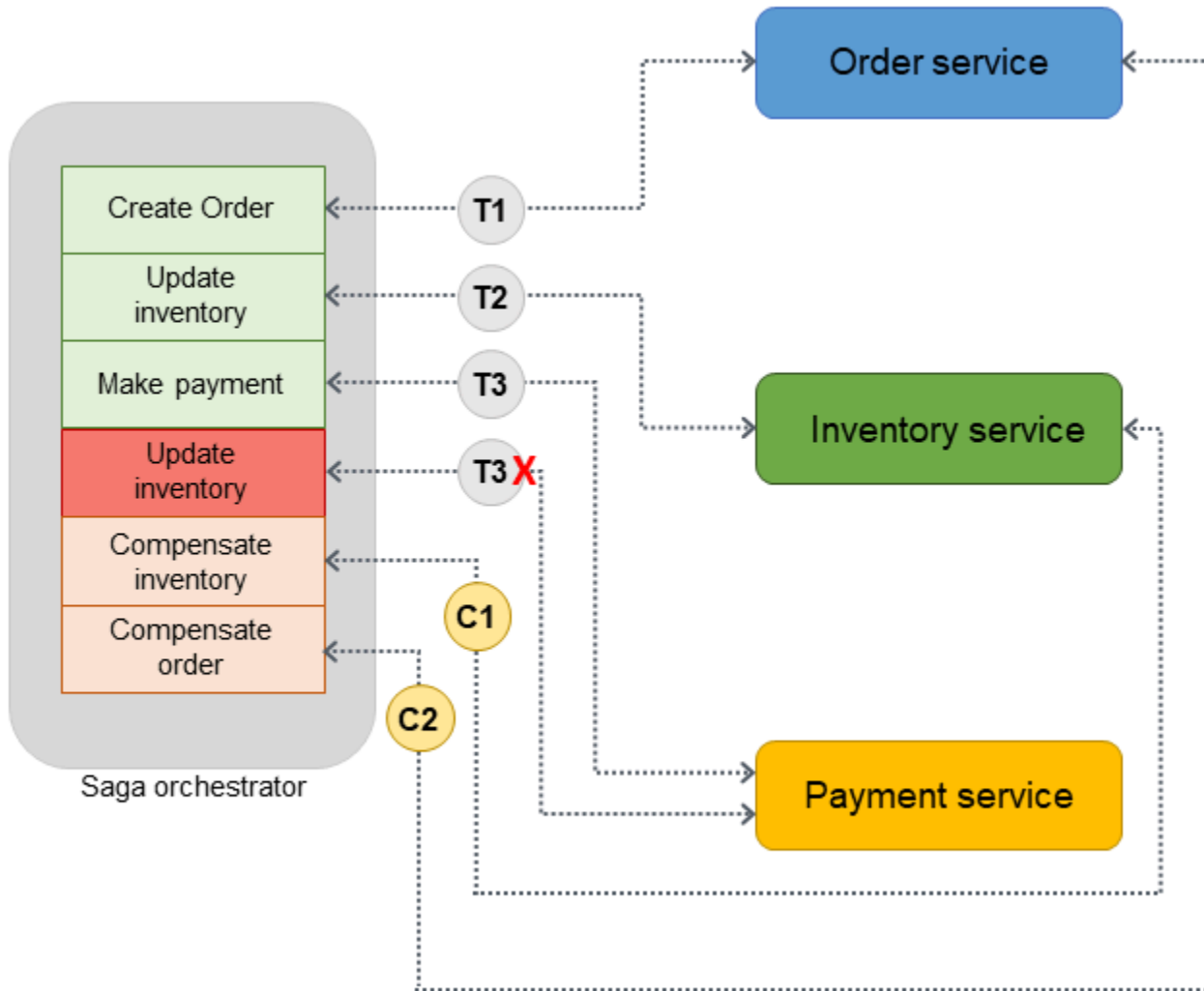
문제 및 고려 사항

- 복잡성: 보정 트랜잭션 및 재시도로 인해 애플리케이션 코드의 복잡성이 가중되어 유지 관리 오버헤드를 초래할 수 있습니다.
- 최종 일관성: 로컬 트랜잭션을 순차적으로 처리하면 최종 일관성이 유지되며, 이는 강력한 일관성이 필요한 시스템에서는 까다로운 과제가 될 수 있습니다. 정합성 모델에 대한 비즈니스 팀의 기대치를 설정하거나 강력한 일관성을 제공하는 데이터베이스로 전환하여 이 문제를 해결할 수 있습니다.
- 멱등성: Saga 참가자는 예상치 못한 충돌 및 오케스트레이터 장애로 인한 일시적 장애 발생 시, 반복 실행을 허용할 수 있는 멱등성을 갖추어야 합니다.
- 트랜잭션 격리: Saga는 트랜잭션 격리를 지원하지 않습니다. 트랜잭션을 동시에 오케스트레이션하면 데이터가 부실해질 수 있습니다. 이러한 시나리오를 처리하려면 시맨틱 잠금을 사용하는 것이 좋습니다.
- 관찰성: 관찰성이란 실행 및 오케스트레이션 프로세스의 문제를 해결하기 위한 상세한 로깅과 추적을 의미합니다. 이는 Saga 참가자 수가 증가하여 디버깅이 복잡해질 경우에 중요해집니다.
- 지연 문제: Saga가 여러 단계로 구성된 경우 보정 트랜잭션으로 인해 전체 응답 시간에서 지연 시간이 늘어날 수 있습니다. 이런 경우에는 동기식 직접 호출을 피하세요.
- 단일 장애 지점: 오케스트레이터는 전체 트랜잭션을 조정하므로 단일 장애 지점이 될 수 있습니다. 이 문제 때문에 Saga 코레오그래피 패턴을 선호하는 경우도 있습니다.

구현

전반적인 아키텍처

다음 아키텍처 다이어그램에서 Saga 오케스트레이터에는 주문 서비스, 인벤토리 서비스, 결제 서비스라는 세 가지 참가자가 참여합니다. 트랜잭션을 완료하려면 T1, T2, T3의 세 단계가 필요합니다. Saga 오케스트레이터는 단계를 인식하고 필요한 순서대로 실행합니다. T3 단계가 실패하면(결제 실패) 오케스트레이터는 보정 트랜잭션 C1과 C2를 실행하여 데이터를 초기 상태로 복원합니다.



트랜잭션이 여러 데이터베이스에 분산되어 있는 경우 [AWS Step Functions](#)를 사용하여 Saga 오케스트레이션을 구현할 수 있습니다.

AWS 서비스를 사용한 구현

이 샘플 솔루션은 Step Functions의 표준 워크플로를 사용하여 Saga 오케스트레이션 패턴을 구현합니다.



고객이 API를 직접적으로 호출하면 Lambda 함수가 간접적으로 호출되고 Lambda 함수에서 전처리가 수행됩니다. 이 함수는 Step Functions 워크플로를 시작하여 분산 트랜잭션 처리를 시작합니다. 전처리가 필요하지 않은 경우 Lambda 함수를 사용하지 않고 API Gateway에서 [직접 Step Functions 워크플로를 시작](#)할 수 있습니다.

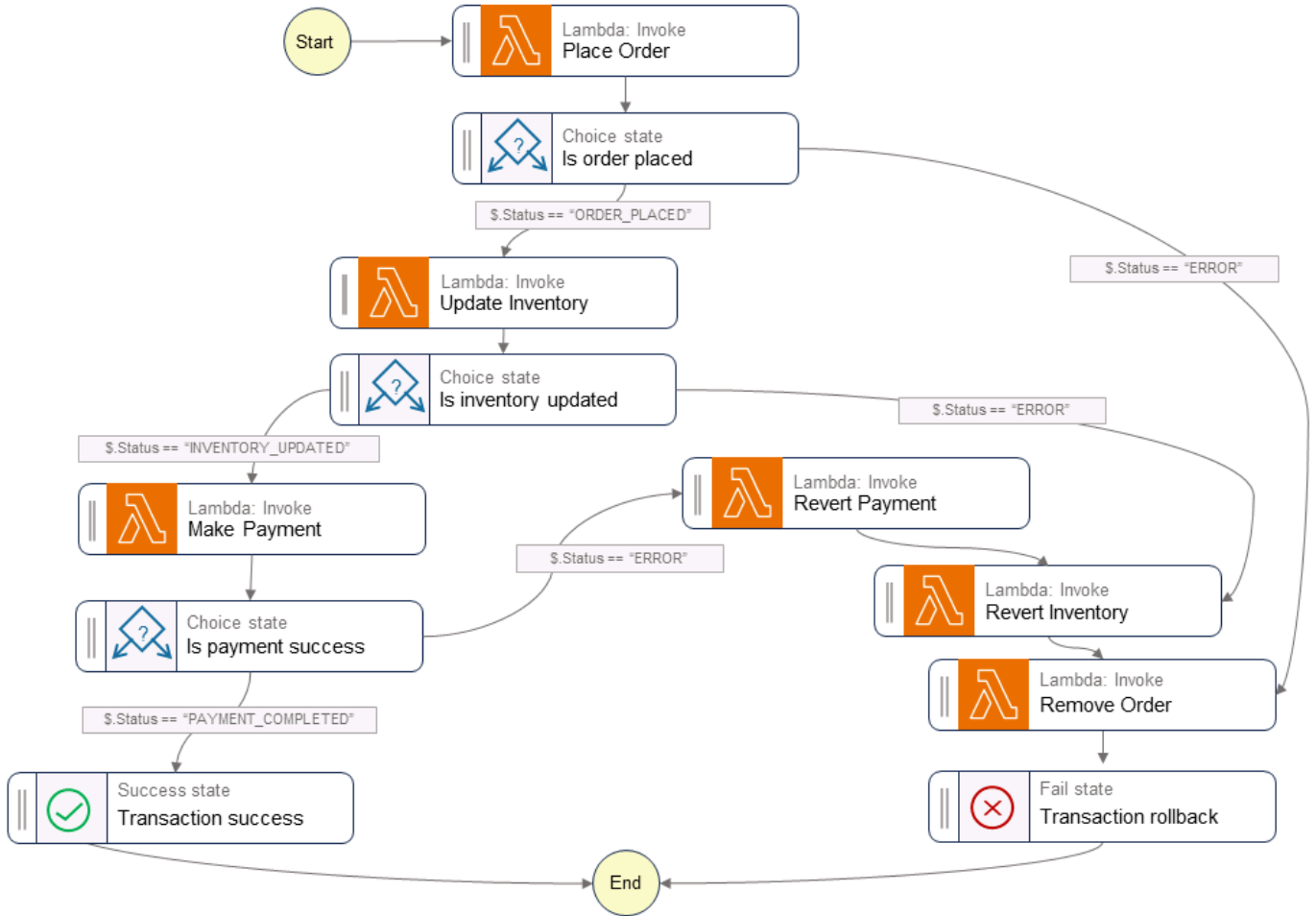
Step Functions를 사용하면 Saga 오케스트레이션 패턴 구현에 내재된 단일 장애 지점 문제를 완화할 수 있습니다. Step Functions는 내결합성을 기본적으로 제공하고 각 AWS 리전의 여러 가용 영역에서 서비스 용량을 유지하여 개별 시스템 또는 데이터 센터 장애로부터 애플리케이션을 보호합니다. 이는 서비스 자체와 서비스가 운영하는 애플리케이션 워크플로 모두의고가용성을 보장하는 데 도움이 됩니다.

Step Functions 워크플로

Step Functions 상태 머신을 사용하면 패턴 구현을 위한 의사 결정 기반 제어 흐름 요구 사항을 구성할 수 있습니다. Step Functions 워크플로는 주문 배치, 재고 업데이트, 결제 처리를 위한 개별 서비스를 직접적으로 호출하여 거래를 완료하고 추가 처리를 위해 이벤트 알림을 보냅니다. Step Functions 워크플로는 트랜잭션을 조정하는 오케스트레이터 역할을 합니다. 워크플로에 오류가 있는 경우 오케스트레이터는 보정 트랜잭션을 실행하여 서비스 전반에서 데이터 무결성이 유지되도록 합니다.

다음 다이어그램은 Step Functions 워크플로에서 실행되는 단계를 보여줍니다. Place Order Update Inventory 및 Make Payment 단계는 성공 경로를 나타냅니다. 주문이 접수되고, 재고가 업데이트되고, 결제가 처리되고, Success 상태가 호출자에게 반환됩니다.

Revert Payment, Revert Inventory 및 Remove Order Lambda 함수는 워크플로의 특정 단계가 실패할 경우 오케스트레이터가 실행하는 보정 트랜잭션을 나타냅니다. 워크플로가 Update Inventory 단계에서 실패하면 오케스트레이터는 Revert Inventory 및 Remove Order 단계를 직접적으로 호출한 후 Fail 상태를 호출자에게 반환합니다. 이러한 보정 트랜잭션은 데이터 무결성을 보장합니다. 재고가 원래 수준으로 돌아가고 주문이 되돌아갑니다.



샘플 코드

다음 샘플 코드는 Step Functions를 사용하여 Saga 오케스트레이터를 생성하는 방법을 보여줍니다. 전체 코드를 보려면 이 예제의 [GitHub리포지토리](#)를 참조하십시오.

태스크 정의

```

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

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

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

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

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

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

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

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

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

Step Functions 및 상태 머신 정의

```
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 리포지토리

이 패턴의 샘플 아키텍처를 완전히 구현하려면 <https://github.com/aws-samples/GitHub> 리포지토리를 참조하십시오 [saga-orchestration-netcore-blog](#).

블로그 참조

- [Building a serverless distributed application using Saga Orchestration pattern](#)

관련 콘텐츠

- [Saga 코레오그래피 패턴](#)
- [트랜잭션 아웃박스 패턴](#)

비디오

다음 비디오에서는 를 사용하여 사가 오케스트레이션 패턴을 구현하는 방법을 설명합니다. AWS Step Functions

스캐터 게더링 패턴

의도

스캐터 게더 패턴은 유사하거나 관련된 요청을 여러 수신자에게 브로드캐스트하고 애그리게이터라는 구성 요소를 사용하여 응답을 다시 단일 메시지로 집계하는 메시지 라우팅 패턴입니다. 이 패턴은 병렬화를 달성하고, 처리 대기 시간을 줄이고, 비동기 통신을 처리하는 데 도움이 됩니다. 동기식 접근 방식을 사용하여 스캐터 게더 패턴을 구현하는 것은 간단하지만 메시징 서비스를 사용하거나 사용하지 않고 비동기 통신에서 메시지 라우팅으로 구현하는 것이 더 강력한 접근 방식입니다.

목적

응용 프로그램 처리에서 순차적으로 처리하는 데 시간이 오래 걸릴 수 있는 요청을 병렬로 처리되는 여러 요청으로 분할할 수 있습니다. API 호출을 통해 여러 외부 시스템에 요청을 보내 응답을 받을 수도 있습니다. 스캐터 게더링 패턴은 여러 소스의 입력이 필요할 때 유용합니다. 스캐터 게더링은 결과를 집계하여 정보에 입각한 결정을 내리거나 요청에 가장 적합한 응답을 선택하는 데 도움이 됩니다.

스캐터 게더링 패턴은 이름에서 알 수 있듯이 두 단계로 구성됩니다.

- 스캐터 단계에서는 요청 메시지를 처리하여 여러 수신자에게 병렬로 보냅니다. 이 단계에서 애플리케이션은 네트워크 전체에 요청을 분산시키고 즉각적인 응답을 기다리지 않고 계속 실행합니다.
- 수집 단계에서 애플리케이션은 수신자로부터 응답을 수집하여 이를 필터링하거나 통합한 응답으로 결합합니다. 모든 응답이 수집되면 이를 단일 응답으로 집계하거나 추후 처리를 위해 가장 적합한 응답을 선택할 수 있습니다.

적용 가능성

스캐터-게더 패턴은 다음과 같은 경우에 사용합니다.

- 다양한 API의 데이터를 집계하고 통합하여 정확한 응답을 만들 계획입니다. 이 패턴은 서로 다른 출처의 정보를 일관된 전체로 통합합니다. 예를 들어 예약 시스템은 여러 수신자에게 여러 외부 파트너로부터 견적을 받도록 요청할 수 있습니다.
- 거래를 완료하려면 동일한 요청을 여러 수신자에게 동시에 보내야 합니다. 예를 들어 이 패턴을 사용하여 인벤토리 데이터를 병렬로 쿼리하여 제품 가용성을 확인할 수 있습니다.

- 요청을 여러 수신자에게 분산하여 부하 분산을 수행할 수 있는 안정적이고 확장 가능한 시스템을 구현하려고 합니다. 한 수신자가 실패하거나 높은 부하를 경험하더라도 다른 수신자는 여전히 요청을 처리할 수 있습니다.
- 여러 데이터 원본이 포함된 복잡한 쿼리를 구현할 때는 성능을 최적화해야 합니다. 쿼리를 관련 데이터베이스로 분산하고 일부 결과를 수집한 다음 종합적인 답변으로 결합할 수 있습니다.
- 샤딩과 복제를 위해 데이터 요청을 여러 데이터 처리 엔드포인트로 라우팅하는 일종의 맵 축소 처리를 구현하고 있습니다. 일부 결과가 필터링되고 결합되어 올바른 응답을 구성합니다.
- 키-값 데이터베이스에서 쓰기가 많은 워크로드의 파티션 키 공간에 쓰기 작업을 분산하려고 합니다. 애그리게이터는 각 샤드의 데이터를 쿼리하여 결과를 읽은 다음 단일 응답으로 통합합니다.

문제 및 고려 사항

- 내결함성: 이 패턴은 병렬로 작동하는 여러 수신자에 의존하므로 실패를 정상적으로 처리하는 것이 중요합니다. 수신자 장애가 전체 시스템에 미치는 영향을 완화하기 위해 이중화, 복제, 장애 감지와 같은 전략을 구현할 수 있습니다.
- 확장 제한: 총 처리 노드 수가 증가하면 관련 네트워크 오버헤드도 증가합니다. 네트워크를 통한 통신과 관련된 모든 요청은 지연 시간을 증가시키고 병렬화의 이점에 부정적인 영향을 미칠 수 있습니다.
- 응답 시간 병목 현상: 최종 처리가 완료되기 전에 모든 수신자를 처리해야 하는 작업의 경우 가장 느린 수신자의 응답 시간으로 인해 전체 시스템의 성능이 제한됩니다.
- 부분 응답: 요청이 여러 수신자에게 분산되어 있는 경우 일부 수신자는 제한 시간이 초과될 수 있습니다. 이러한 경우 구현에서 클라이언트에게 응답이 불완전하다는 사실을 전달해야 합니다. UI 프론트엔드를 사용하여 응답 집계 세부 정보를 표시할 수도 있습니다.
- 데이터 일관성: 여러 수신자의 데이터를 처리하는 경우 최종 집계 결과가 정확하고 일관되도록 데이터 동기화 및 충돌 해결 기술을 신중하게 고려해야 합니다.

구현

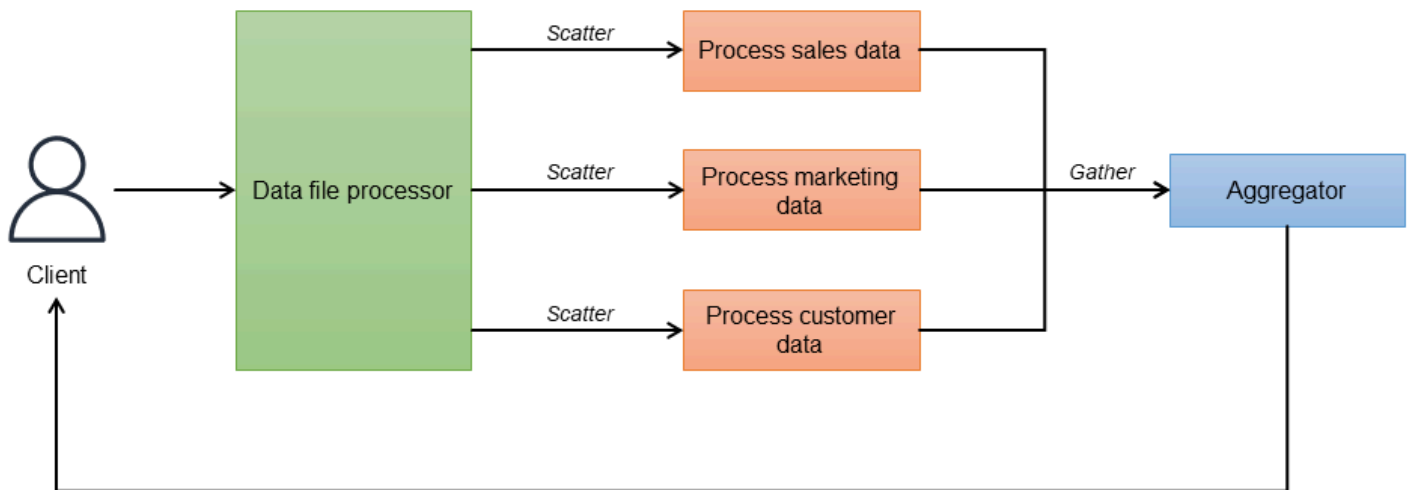
전반적인 아키텍처

스캐터 게더 패턴은 루트 컨트롤러를 사용하여 요청을 처리할 수신자에게 요청을 배포하는 데 사용됩니다. 스캐터 단계에서 이 패턴은 두 가지 메커니즘을 사용하여 수신자에게 메시지를 보낼 수 있습니다.

- 배포별 분산: 응용 프로그램에는 결과를 얻기 위해 호출해야 하는 알려진 수신자 목록이 있습니다. 수신자는 고유한 기능을 가진 서로 다른 프로세스일 수도 있고 처리 부하를 분산하기 위해 확장된 단일 프로세스일 수도 있습니다. 처리 노드 중 하나라도 제한 시간이 초과되거나 응답이 지연되는 경우 컨트롤러는 처리를 다른 노드로 재배포할 수 있습니다.
- 스캐터 바이 옥션: 애플리케이션은 게시-구독 패턴을 사용하여 관심 있는 수신자에게 메시지를 브로드캐스트합니다. 이 경우 수신자는 언제든지 메시지를 구독하거나 구독을 철회할 수 있습니다.

배포에 따라 분산됩니다.

분산형 분산 방식에서는 루트 컨트롤러가 들어오는 요청을 독립된 작업으로 나누어 사용 가능한 수신자에게 할당합니다 (분산 단계). 각 수신자 (프로세스, 컨테이너 또는 Lambda 함수) 는 독립적으로 병렬 계산을 수행하며 응답의 일부를 생성합니다. 수신자는 작업을 완료하면 애그리게이터에 응답을 보냅니다 (수집 단계). 애그리게이터는 부분 응답을 결합하여 최종 결과를 클라이언트에 반환합니다. 다음 다이어그램은 이 워크플로를 보여줍니다.

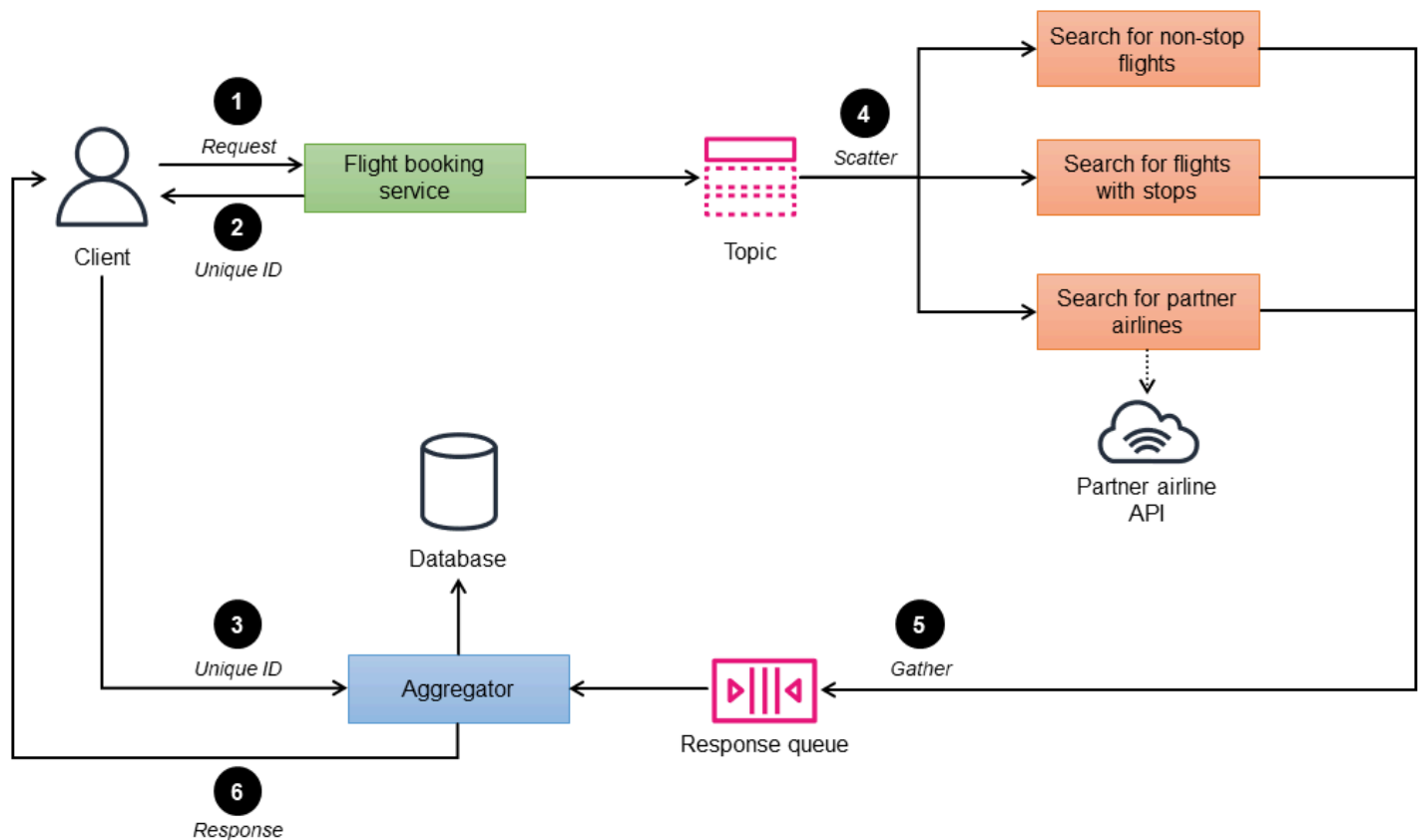


컨트롤러 (데이터 파일 프로세서) 는 전체 호출 세트를 조정하고 호출할 모든 예약 엔드포인트를 인식합니다. 너무 오래 걸리는 응답을 무시하도록 타임아웃 파라미터를 구성할 수 있습니다. 요청이 전송되면 애그리게이터는 각 엔드포인트에서 응답이 돌아올 때까지 기다립니다. 복원력을 구현하기 위해 각 마이크로서비스를 여러 인스턴스와 함께 배포하여 부하 분산을 수행할 수 있습니다. 애그리게이터는 결과를 가져와 단일 응답 메시지로 결합하고 추가 처리 전에 중복 데이터를 제거합니다. 제한 시간이 초과된 응답은 무시됩니다. 컨트롤러는 별도의 애그리게이터 서비스를 사용하는 대신 애그리게이터 역할을 할 수도 있습니다.

경매를 통한 스캐터

컨트롤러가 수신자를 인식하지 못하거나 수신자가 서로 느슨하게 연결되어 있는 경우 스캐터 바이 옥션 방법을 사용할 수 있습니다. 이 방법에서는 수신자가 주제를 구독하고 컨트롤러는 요청을 주제에 게시합니다. 수신자는 결과를 응답 대기열에 게시합니다. 루트 컨트롤러가 수신자를 인식하지 못하기 때문에 수집 프로세스에서는 애그리게이터 (또 다른 메시징 패턴) 를 사용하여 응답을 수집하고 이를 단일 응답 메시지로 추출합니다. 애그리게이터는 고유한 ID를 사용하여 요청 그룹을 식별합니다.

예를 들어 다음 다이어그램에서 스캐터 바이 옥션 방법은 항공사 웹 사이트의 항공편 예약 서비스를 구현하는 데 사용됩니다. 웹 사이트에서는 사용자가 항공사 자체 항공사 및 파트너 항공사의 항공편을 검색하고 표시할 수 있으며 검색 상태를 실시간으로 표시해야 합니다. 항공편 예약 서비스는 논스톱 항공편, 경유지가 있는 항공편, 제휴 항공사의 세 가지 검색 마이크로서비스로 구성됩니다. 파트너 항공사 검색은 파트너의 API 엔드포인트를 호출하여 응답을 받습니다.



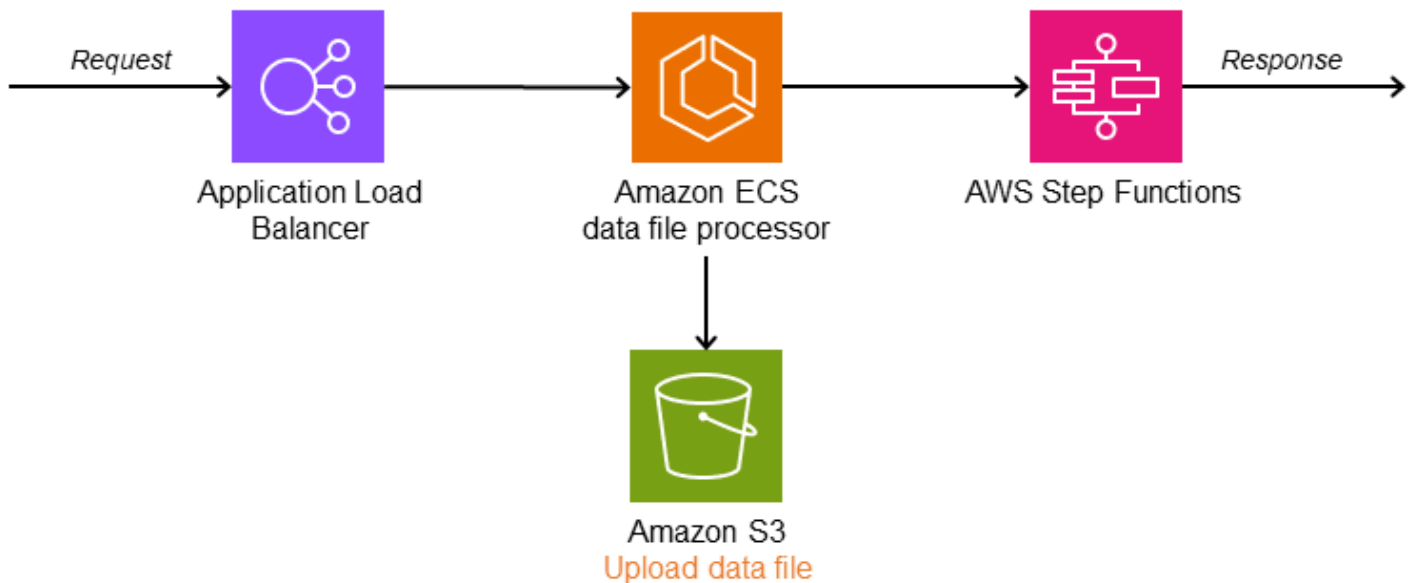
1. 항공편 예약 서비스 (컨트롤러) 는 클라이언트로부터 검색 기준을 입력받아 요청을 처리하고 주제에 게시합니다.
2. 컨트롤러는 고유 ID를 사용하여 각 요청 그룹을 식별합니다.
3. 클라이언트는 6단계에서 애그리게이터에 고유 ID를 보냅니다.

4. 예약 주제를 구독한 예약 검색 마이크로서비스가 요청을 수신합니다.
5. 마이크로서비스는 요청을 처리하고 지정된 검색 기준에 맞는 좌석 가용성을 응답 대기열에 반환합니다.
6. 애그리게이터는 임시 데이터베이스에 저장된 모든 응답 메시지를 수집하고, 고유한 ID별로 항공편을 그룹화하고, 단일 통합 응답을 생성하여 클라이언트로 다시 보냅니다.

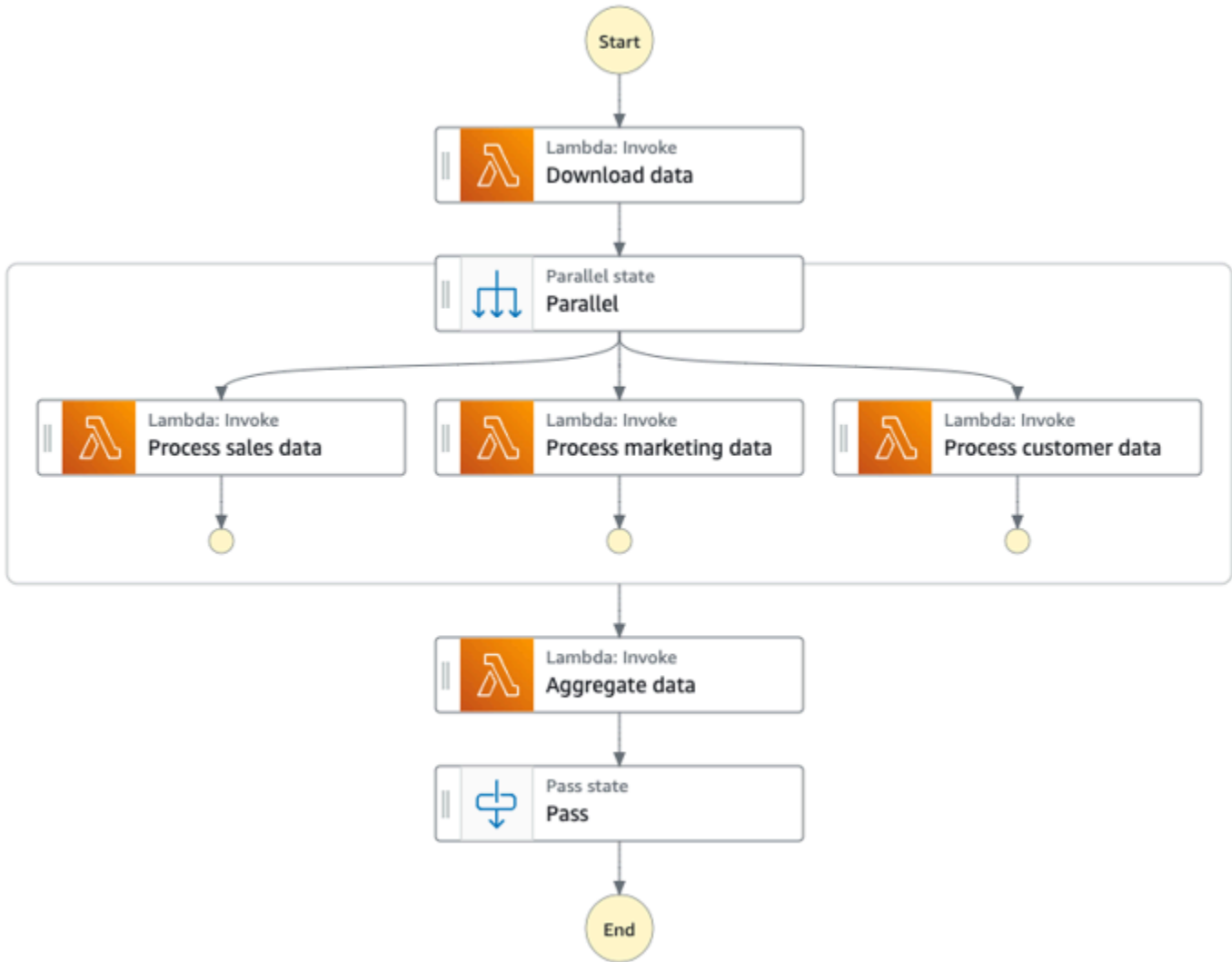
구현은 다음을 사용합니다. AWS 서비스

배포에 따른 스캐터

다음 아키텍처에서 루트 컨트롤러는 수신 요청 데이터를 개별 Amazon Simple Storage Service (Amazon S3) 버킷으로 분할하고 워크플로를 시작하는 데이터 파일 프로세서 (Amazon ECS)입니다. AWS Step Functions 워크플로우는 데이터를 다운로드하고 병렬 파일 처리를 시작합니다. Parallel상태는 모든 태스크가 응답을 반환할 때까지 기다립니다. AWS Lambda 함수는 데이터를 집계하여 Amazon S3에 다시 저장합니다.

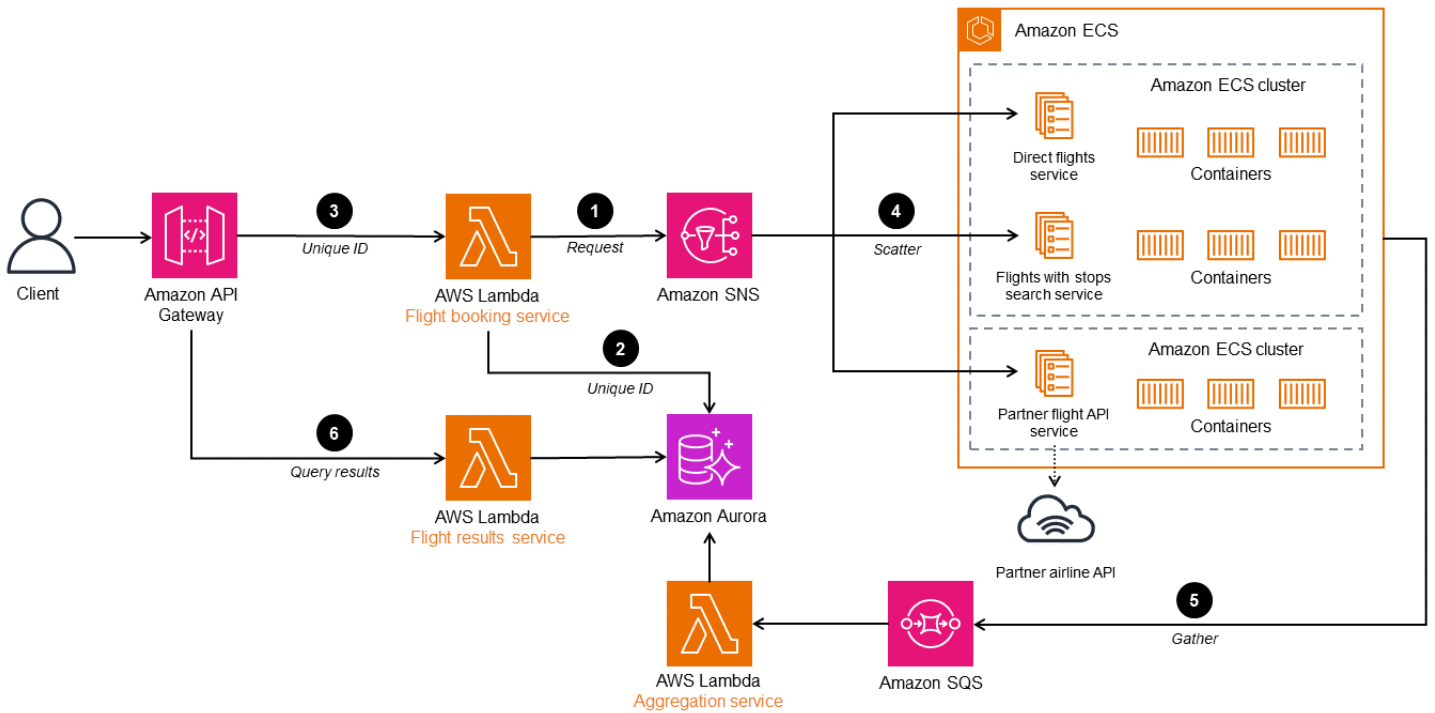


다음 다이어그램은 Parallel 상태를 포함한 Step Functions 워크플로를 보여줍니다.



경매를 통한 스캐터

다음 다이어그램은 경매 방법별 스캐터 AWS 아키텍처를 보여줍니다. 루트 컨트롤러 항공편 예약 서비스는 항공편 검색 요청을 여러 마이크로서비스로 분산합니다. 게시-구독 채널은 커뮤니케이션을 위한 관리형 메시징 서비스인 Amazon Simple Notification Service (Amazon SNS) 를 통해 구현됩니다. Amazon SNS는 분리된 마이크로서비스 애플리케이션 간의 메시지 또는 사용자와의 직접 통신을 지원합니다. 더 나은 관리와 확장성을 위해 Amazon Elastic Kubernetes Service (Amazon EKS) 또는 Amazon Elastic Container Service (Amazon ECS) 에 수신자 마이크로서비스를 배포할 수 있습니다. 비행 결과 서비스는 결과를 클라이언트에 반환합니다. Amazon ECS AWS Lambda 또는 Amazon EKS 와 같은 다른 컨테이너 오케스트레이션 서비스 또는 기타 컨테이너 오케스트레이션 서비스에서 구현할 수 있습니다.



1. 항공편 예약 서비스 (컨트롤러) 는 클라이언트로부터 검색 기준을 입력받아 요청을 처리하고 SNS 주제에 게시합니다.
2. 컨트롤러는 요청을 식별하기 위해 Amazon Aurora 데이터베이스에 고유 ID를 게시합니다.
3. 클라이언트는 6단계에서 클라이언트에 고유 ID를 보냅니다.
4. 예약 주제를 구독한 예약 검색 마이크로서비스가 요청을 수신합니다.
5. 마이크로서비스는 요청을 처리하고 지정된 검색 기준에 대한 좌석 가용성을 Amazon Simple Queue Service (Amazon SQS) 의 응답 대기열로 반환합니다. 애그리게이터는 모든 응답 메시지를 대조하여 임시 데이터베이스에 저장합니다.
6. 비행 결과 서비스는 항공편을 고유 ID별로 그룹화하고 단일 통합 응답을 생성하여 클라이언트로 다시 전송합니다.

이 아키텍처에 다른 항공사 검색을 추가하려면 SNS 주제를 구독하고 SQS 대기열에 게시하는 마이크로서비스를 추가하면 됩니다.

요약하자면 분산 시스템은 스캐터 게더링 패턴을 통해 효율적인 병렬화를 달성하고 지연 시간을 줄이며 비동기 통신을 원활하게 처리할 수 있습니다.

GitHub 리포지토리

이 패턴의 샘플 아키텍처를 완전히 구현하려면 <https://github.com/aws-samples/GitHub-asynchronous-messaging-workshop/tree/master/code/lab-3>의 리포지토리를 참조하십시오.

워크숍

- [분리된](#) 마이크로서비스 워크숍의 스캐터 게더링 랩

블로그 참조

- [마이크로서비스의 애플리케이션 통합 패턴](#)

관련 콘텐츠

- [퍼블리시-서브스크립션](#) 패턴

스트랭글러 무화과 패턴

Intent

스트랭글러 피그 패턴은 변환 위험과 비즈니스 중단을 줄이면서 모놀리식 애플리케이션을 마이크로서비스 아키텍처로 점진적으로 마이그레이션하는 데 도움이 됩니다.

동기 부여

모놀리식 애플리케이션은 단일 프로세스 또는 컨테이너 내에서 대부분의 기능을 제공하도록 개발되었습니다. 코드는 밀접하게 결합되어 있습니다. 따라서 애플리케이션 변경 시 회귀 문제를 방지하려면 철저한 재테스트가 필요합니다. 변경 사항은 개별적으로 테스트할 수 없으므로 주기 시간에 영향을 미칩니다. 애플리케이션에 더 많은 기능이 추가됨에 따라 복잡성이 높으면 유지 관리에 소요되는 시간이 늘어나고 출시 시간이 길어지며 결과적으로 제품 혁신이 느려질 수 있습니다.

애플리케이션 규모가 확장되면 팀의 인지 부하가 증가하고 팀 소유권 경계가 불분명해질 수 있습니다. 부하에 따라 개별 기능을 확장할 수는 없습니다. 최대 부하를 지원하도록 전체 애플리케이션을 확장해야 합니다. 시스템이 노후화되면 기술이 더 이상 사용되지 않아 지원 비용이 증가할 수 있습니다. 모놀리식 레거시 애플리케이션은 개발 당시 이용 가능했지만 배포용으로 설계되지 않은 모범 사례를 따릅니다.

모놀리식 애플리케이션을 마이크로서비스 아키텍처로 마이그레이션하면 더 작은 구성 요소로 분할할 수 있습니다. 이러한 구성 요소는 독립적으로 확장할 수 있고, 독립적으로 릴리스할 수 있으며, 개별 팀이 소유할 수 있습니다. 변경 사항이 현지화되고 신속하게 테스트 및 릴리스할 수 있기 때문에 변경 속도가 빨라집니다. 구성 요소가 느슨하게 결합되고 개별적으로 배포될 수 있기 때문에 변경 사항이 미치는 영향 범위가 더 작습니다.

코드를 재작성하거나 리팩토링하여 모놀리식을 마이크로서비스 애플리케이션으로 완전히 대체하는 것은 엄청난 작업이며 위험도 큼니다. 단일 작업으로 모놀리식을 마이그레이션하는 빅뱅 마이그레이션은 변환 위험과 비즈니스 중단을 초래합니다. 애플리케이션을 리팩토링하는 동안에는 새 기능을 추가하기가 매우 어렵거나 심지어 불가능하기도 합니다.

이 문제를 해결하는 한 가지 방법은 Martin Fowler가 도입한 스트랭글러 무화과 패턴을 사용하는 것입니다. 이 패턴에는 기능을 점진적으로 추출하여 마이크로서비스로 이동하고 기존 시스템을 중심으로 새 애플리케이션을 만드는 것이 포함됩니다. 모놀리식의 기능은 점차 마이크로서비스로 대체되고 있으며, 애플리케이션 사용자는 새로 마이그레이션된 기능을 점진적으로 사용할 수 있습니다. 모든 기능을 새 시스템으로 옮기면 모놀리식 애플리케이션을 안전하게 폐기할 수 있습니다.

적용 가능성

스트랭글러 무화과 패턴을 사용하는 경우는 다음과 같습니다.

- 모놀리식 애플리케이션을 마이크로서비스 아키텍처로 점진적으로 마이그레이션하고 싶을 것입니다.
- 빅뱅 마이그레이션 접근 방식은 모놀리스의 크기와 복잡성 때문에 위험합니다.
- 기업은 새로운 기능을 추가하기를 원하며 전환이 완료될 때까지 기다릴 수 없습니다.
- 변환 과정에서 최종 사용자에게 미치는 영향을 최소화해야 합니다.

문제 및 고려 사항

- 코드 기반 액세스: 스트랭글러 fig 패턴을 구현하려면 모놀리식 애플리케이션의 코드 베이스에 액세스할 수 있어야 합니다. 기능이 모놀리스 외부로 마이그레이션되면 코드를 약간 변경하고 모놀리스 내에 손상 방지 계층을 구현하여 호출을 새 마이크로서비스로 라우팅해야 합니다. 코드베이스 액세스 없이는 통화를 가로챌 수 없습니다. 코드 기반 액세스는 들어오는 요청을 리디렉션할 때도 중요합니다. 프록시 계층이 마이그레이션된 기능에 대한 호출을 가로채서 마이크로서비스로 라우팅하려면 일부 코드 리팩토링이 필요할 수 있습니다.
- 불분명한 도메인: 시스템을 조기에 분해하면 비용이 많이 들 수 있으며, 특히 도메인이 명확하지 않은 경우 비용이 많이 들 수 있으며 서비스 경계를 잘못 설정할 수 있습니다. 도메인 기반 설계 (DDD)는 도메인을 이해하기 위한 메커니즘이고 이벤트 스토밍은 도메인 경계를 결정하는 기법입니다.
- 마이크로서비스 식별: DDD를 마이크로서비스를 식별하는 핵심 도구로 사용할 수 있습니다. 마이크로서비스를 식별하려면 서비스 클래스 간의 자연스러운 구분을 살펴보세요. 많은 서비스가 자체 데이터 액세스 객체를 소유하며 쉽게 분리됩니다. 종속성이 없거나 거의 없는 관련 비즈니스 로직 및 클래스가 있는 서비스는 마이크로서비스에 적합합니다. 모놀리스를 분해하기 전에 코드를 리팩토링하여 긴밀한 결합을 방지할 수 있습니다. 또한 규정 준수 요구 사항, 릴리스 주기, 팀의 지리적 위치, 규모 조정 요구 사항, 사용 사례 기반 기술 요구 사항, 팀의 인지 부하도 고려해야 합니다.
- 손상 방지 계층: 마이그레이션 프로세스 중에 모놀리스 내의 기능이 마이크로서비스로 마이그레이션된 기능을 호출해야 하는 경우 각 호출을 적절한 마이크로서비스로 라우팅하는 손상 방지 계층 (ACL)을 구현해야 합니다. 모놀리스 내에서 기존 발신자를 분리하고 변경을 방지하기 위해 ACL은 호출을 최신 인터페이스로 변환하는 어댑터 또는 파사드 역할을 합니다. 이에 대해서는 이 가이드 앞 부분에 있는 ACL 패턴의 [구현 섹션에서](#) 자세히 설명합니다.
- 프록시 계층 장애: 마이그레이션 중에 프록시 계층은 모놀리식 애플리케이션으로 이동하는 요청을 가로채서 기존 시스템이나 새 시스템으로 라우팅합니다. 하지만 이 프록시 계층은 단일 장애 지점이나 성능 병목 현상이 될 수 있습니다.

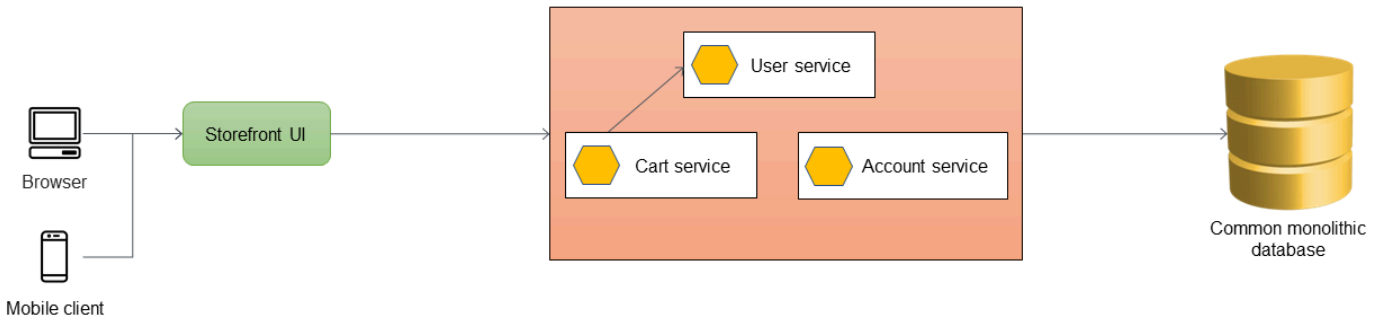
- 애플리케이션 복잡성: 대형 모놀리식은 스트랭글러 피그 패턴의 이점을 가장 많이 누릴 수 있습니다. 전체 리팩토링의 복잡성이 낮은 소규모 애플리케이션의 경우 마이그레이션하는 대신 마이크로서비스 아키텍처에서 애플리케이션을 다시 작성하는 것이 더 효율적일 수 있습니다.
- 서비스 상호 작용: 마이크로서비스는 동기식 또는 비동기식으로 통신할 수 있습니다. 동기 통신이 필요한 경우 시간 초과로 인해 연결 또는 스레드 풀 소비가 발생하여 애플리케이션 성능 문제가 발생할 수 있는지 고려하세요. 이러한 경우에는 [회로 차단기 패턴](#)을 사용하여 장기간 실패할 가능성이 있는 작업에 대해 즉시 실패를 반환하십시오. 이벤트 및 메시징 대기열을 사용하여 비동기 통신을 수행할 수 있습니다.
- 데이터 집계: 마이크로서비스 아키텍처에서는 데이터가 데이터베이스 전체에 분산됩니다. 데이터 집계가 필요한 경우 프론트 [AWS AppSync](#) 엔드에서 사용하거나 백엔드의 CQRS (명령 쿼리 책임 분리) 패턴을 사용할 수 있습니다.
- 데이터 일관성: 마이크로서비스는 자체 데이터 저장소를 소유하며, 모놀리식 애플리케이션도 이 데이터를 잠재적으로 사용할 수 있습니다. 공유를 활성화하려면 대기열 및 에이전트를 사용하여 새 마이크로서비스의 데이터 저장소를 모놀리식 애플리케이션의 데이터베이스와 동기화할 수 있습니다. 그러나 이로 인해 두 데이터 저장소 간에 데이터 중복과 최종 일관성이 발생할 수 있으므로 데이터 레이크와 같은 장기적인 솔루션을 구축할 수 있을 때까지는 전술적 솔루션으로 취급하는 것이 좋습니다.

구현

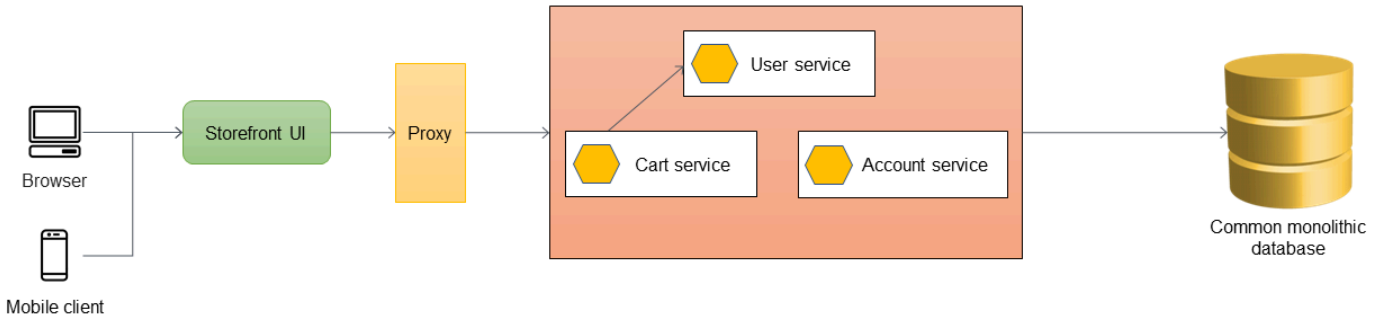
스트랭글러 그림 패턴에서는 특정 기능을 한 번에 하나의 구성 요소씩 새 서비스나 애플리케이션으로 대체합니다. 프록시 계층은 모놀리식 애플리케이션으로 이동하는 요청을 가로채서 레거시 시스템이나 새 시스템으로 라우팅합니다. 프록시 계층은 사용자를 올바른 애플리케이션으로 라우팅하므로 모놀리스가 계속 작동하도록 하면서 새 시스템에 기능을 추가할 수 있습니다. 새 시스템은 결국 기존 시스템의 모든 기능을 대체하므로 사용을 중지할 수 있습니다.

고수준 아키텍처

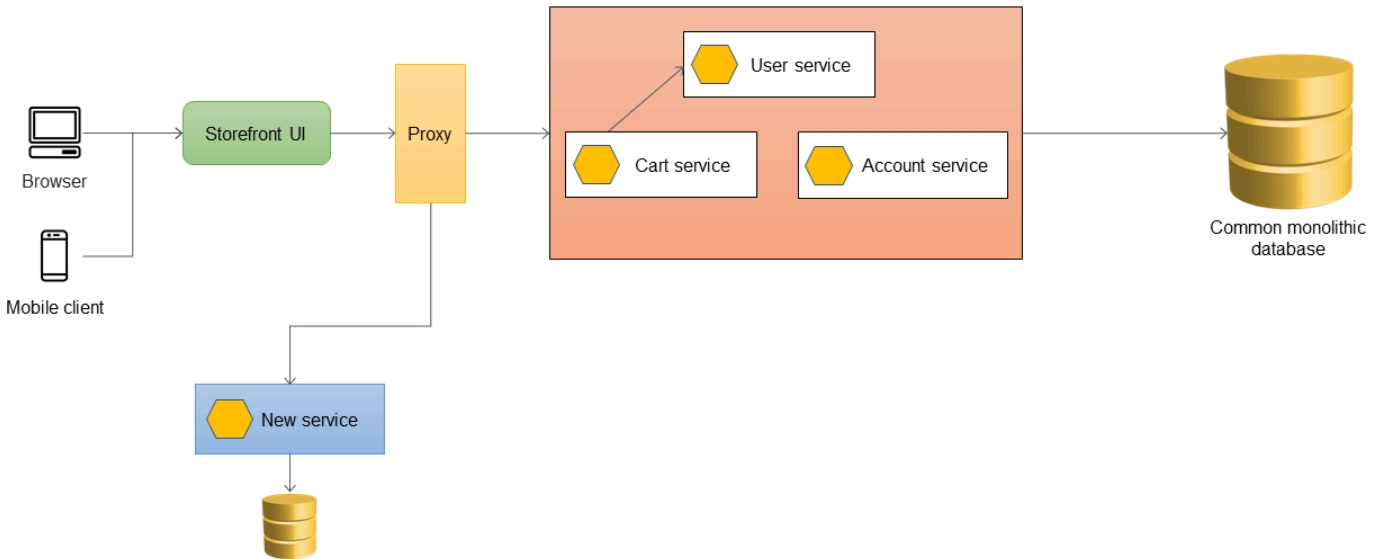
다음 다이어그램에서 모놀리식 애플리케이션에는 사용자 서비스, 카트 서비스, 계정 서비스라는 세 가지 서비스가 있습니다. 카트 서비스는 사용자 서비스에 따라 달라지며 애플리케이션은 모놀리식 관계형 데이터베이스를 사용합니다.



첫 번째 단계는 스토어프론트 UI와 모놀리식 애플리케이션 사이에 프록시 레이어를 추가하는 것입니다. 처음에는 프록시가 모든 트래픽을 모놀리식 애플리케이션으로 라우팅합니다.

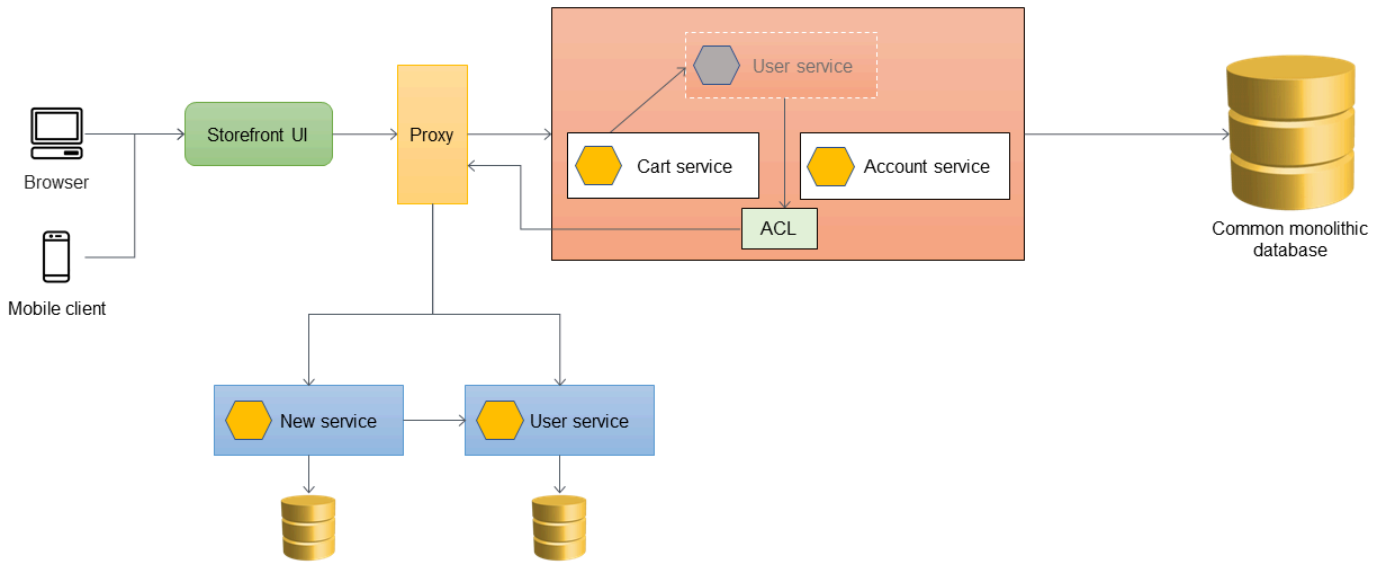


애플리케이션에 새 기능을 추가하려면 기존 모놀리스에 기능을 추가하는 대신 새 마이크로서비스로 구현해야 합니다. 하지만 애플리케이션 안정성을 보장하기 위해 계속해서 모놀리스의 버그를 수정합니다. 다음 다이어그램에서 프록시 계층은 API URL을 기반으로 호출을 모놀리스 또는 새 마이크로서비스로 라우팅합니다.



손상 방지 계층 추가

다음 아키텍처에서는 사용자 서비스가 마이크로서비스로 마이그레이션되었습니다. 카트 서비스가 사용자 서비스를 호출하지만 모놀리스 내에서는 더 이상 구현을 사용할 수 없습니다. 또한 새로 마이그레이션된 서비스의 인터페이스가 모놀리식 애플리케이션 내의 이전 인터페이스와 일치하지 않을 수 있습니다. 이러한 변경 사항을 해결하려면 ACL을 구현해야 합니다. 마이그레이션 프로세스 중에 모놀리스 내의 기능이 마이크로서비스로 마이그레이션된 기능을 호출해야 하는 경우 ACL은 호출을 새 인터페이스로 변환하여 적절한 마이크로서비스로 라우팅합니다.



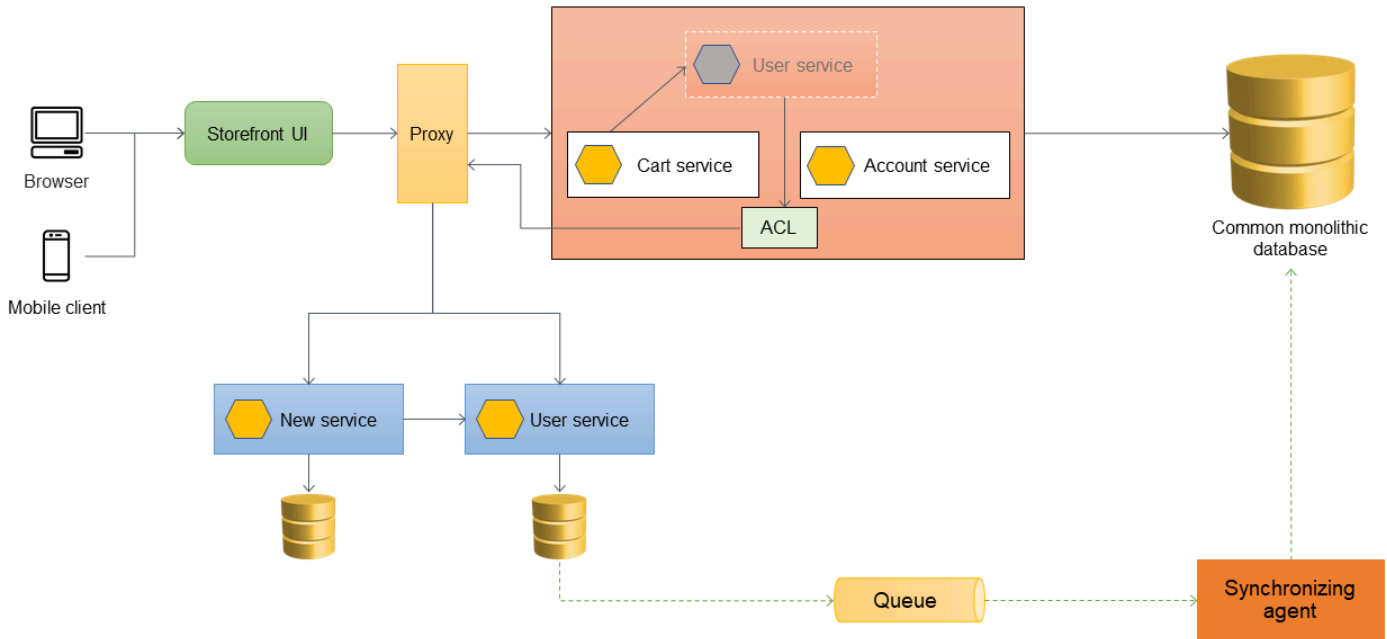
마이그레이션된 서비스와 관련된 클래스 (예: 또는) 로 모놀리식 애플리케이션 내에서 ACL을 구현할 수 있습니다. `UserServiceFacade` `UserServiceAdapter` 모든 종속 서비스를 마이크로서비스 아키텍처로 마이그레이션한 후에는 ACL을 사용 중지해야 합니다.

ACL을 사용하는 경우 카트 서비스는 여전히 모놀리스 내에서 사용자 서비스를 호출하고 사용자 서비스는 ACL을 통해 호출을 마이크로서비스로 리디렉션합니다. 카트 서비스는 마이크로서비스 마이그레이션을 인식하지 않고도 여전히 사용자 서비스를 호출해야 합니다. 회귀 현상과 비즈니스 중단을 줄이려면 이러한 느슨한 결합이 필요합니다.

데이터 동기화 처리

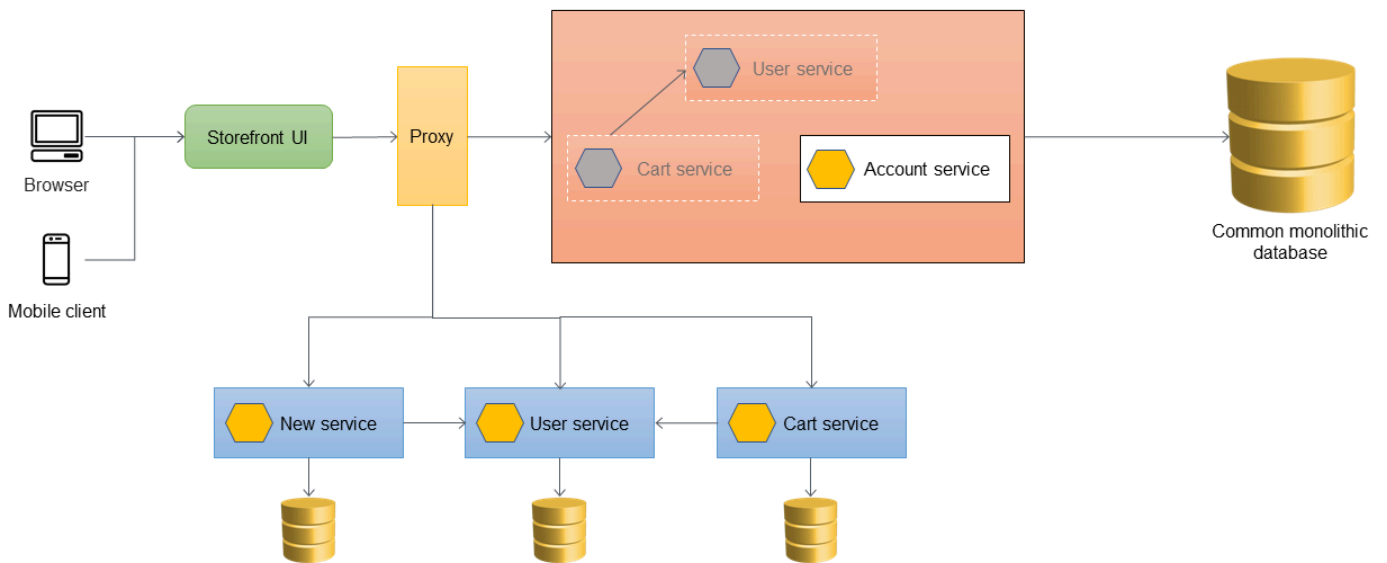
가장 좋은 방법은 마이크로서비스가 자체 데이터를 소유하는 것입니다. 사용자 서비스는 자체 데이터 스토어에 데이터를 저장합니다. 보고와 같은 종속성을 처리하고 아직 마이크로서비스에 직접 액세스할 준비가 되지 않은 다운스트림 애플리케이션을 지원하려면 데이터를 모놀리식 데이터베이스와 동기화해야 할 수 있습니다. 모놀리식 애플리케이션에는 아직 마이크로서비스로 마이그레이션되지 않은 다른 기능 및 구성 요소에 대한 데이터가 필요할 수도 있습니다. 따라서 새 마이크로서비스와 모놀리스 간에는 데이터 동기화가 필요합니다. 데이터를 동기화하려면 다음 다이어그램과 같이 사용자 마이크

로서비스와 모놀리식 데이터베이스 간에 동기화 에이전트를 도입할 수 있습니다. 사용자 마이크로서비스는 데이터베이스가 업데이트될 때마다 이벤트를 대기열에 보냅니다. 동기화 에이전트는 대기열을 수신하고 모놀리식 데이터베이스를 지속적으로 업데이트합니다. 모놀리식 데이터베이스의 데이터는 결국 동기화 중인 데이터와 일치하게 됩니다.

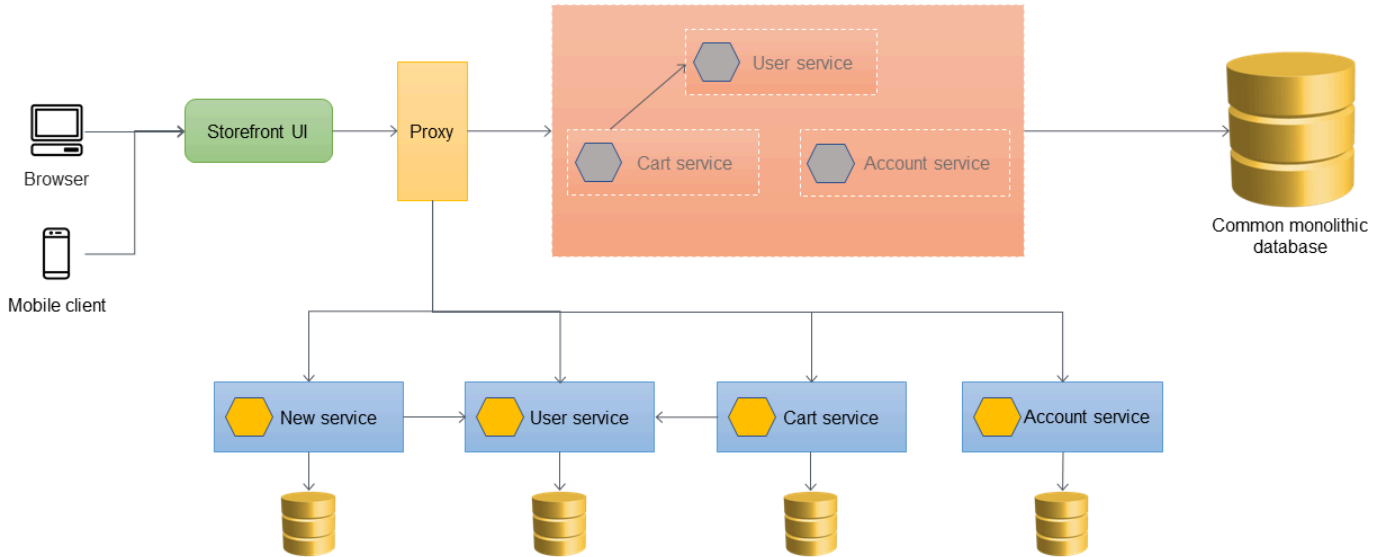


추가 서비스 마이그레이션

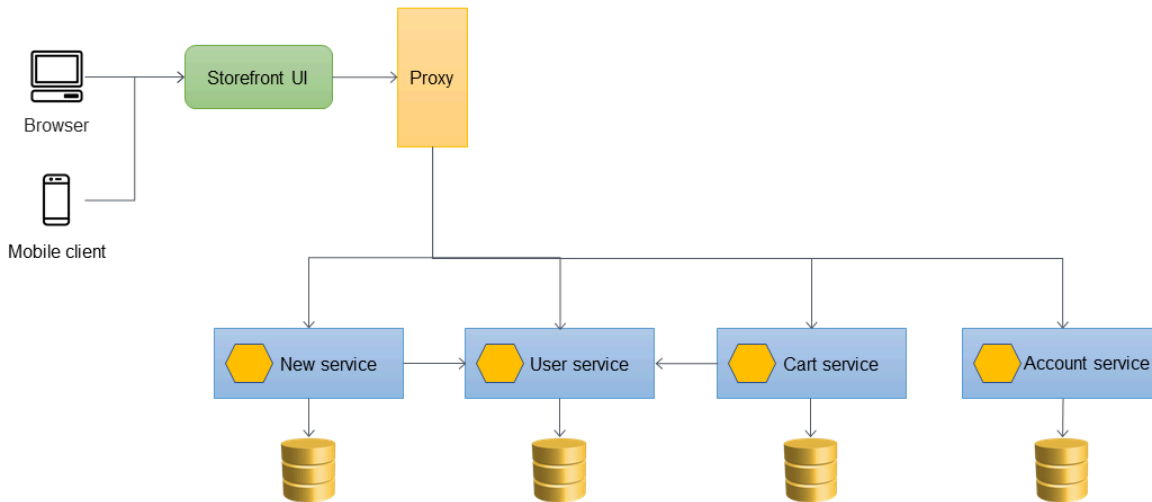
카트 서비스가 모놀리식 애플리케이션 외부로 마이그레이션되면 새 서비스를 직접 호출하도록 코드가 수정되므로 ACL은 더 이상 해당 호출을 라우팅하지 않습니다. 다음은 이러한 아키텍처를 나타낸 다이어그램입니다.



다음 다이어그램은 모든 서비스가 모놀리스 밖으로 마이그레이션되고 모놀리스의 골격만 남아 있는 최종 교란 상태를 보여줍니다. 기록 데이터는 개별 서비스가 소유한 데이터 저장소로 마이그레이션할 수 있습니다. ACL을 제거할 수 있으며, 이 단계에서 모놀리스를 폐기할 준비가 되었습니다.



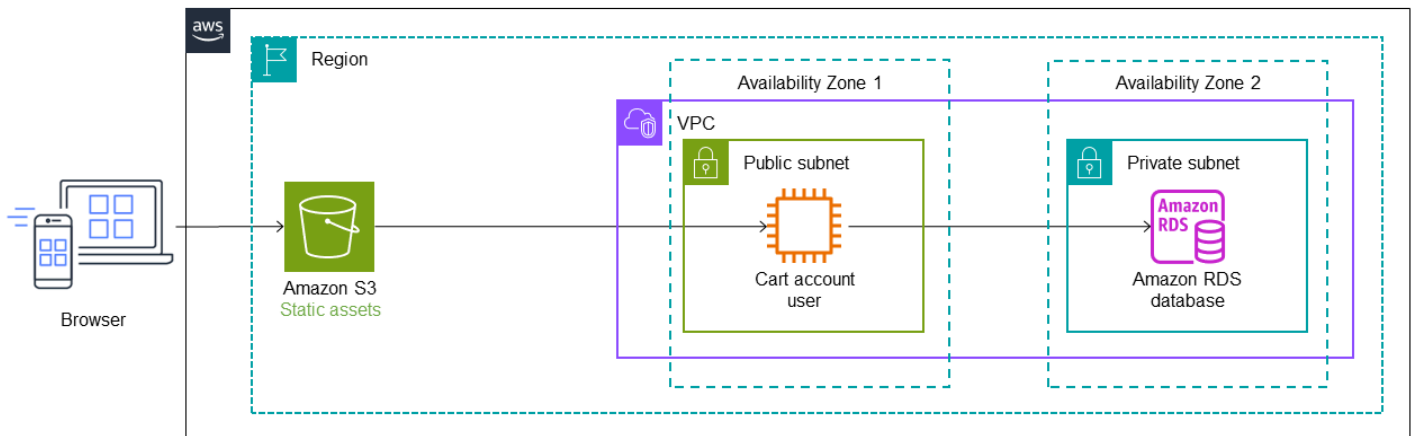
다음 다이어그램은 모놀리식 애플리케이션이 폐기된 후의 최종 아키텍처를 보여줍니다. 애플리케이션 요구 사항에 따라 리소스 기반 URL (예:http://www.storefront.com/user) 또는 자체 도메인 (예:) 을 통해 개별 마이크로서비스를 호스팅할 수 있습니다. [http://user.storefront.com](#) 호스트 이름과 경로를 사용하여 업스트림 소비자에게 HTTP API를 노출하는 주요 방법에 대한 자세한 내용은 API 라우팅 패턴 섹션을 참조하십시오.



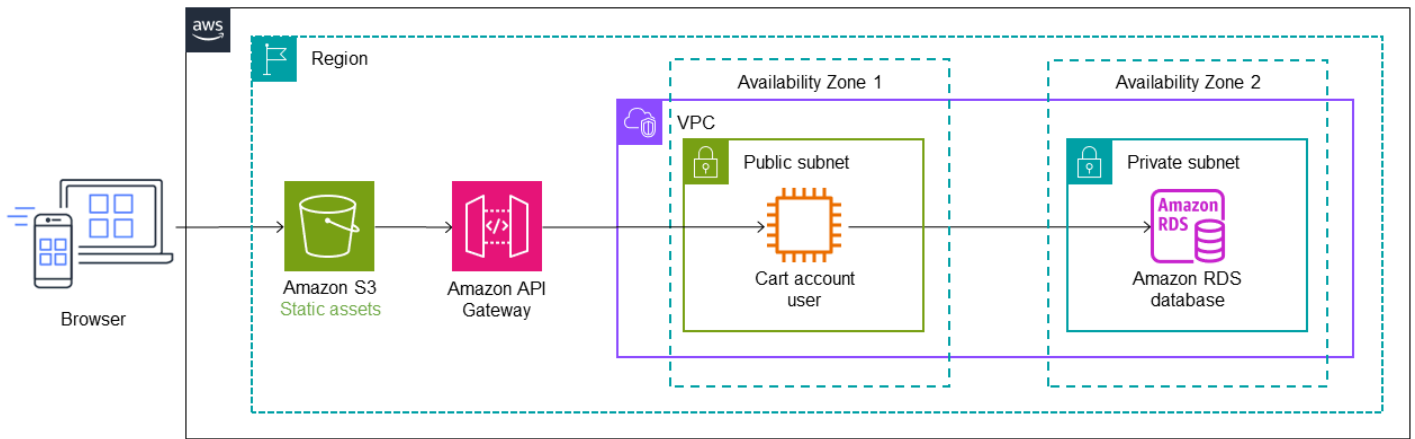
AWS서비스를 사용한 구현

API Gateway를 애플리케이션 프록시로 사용

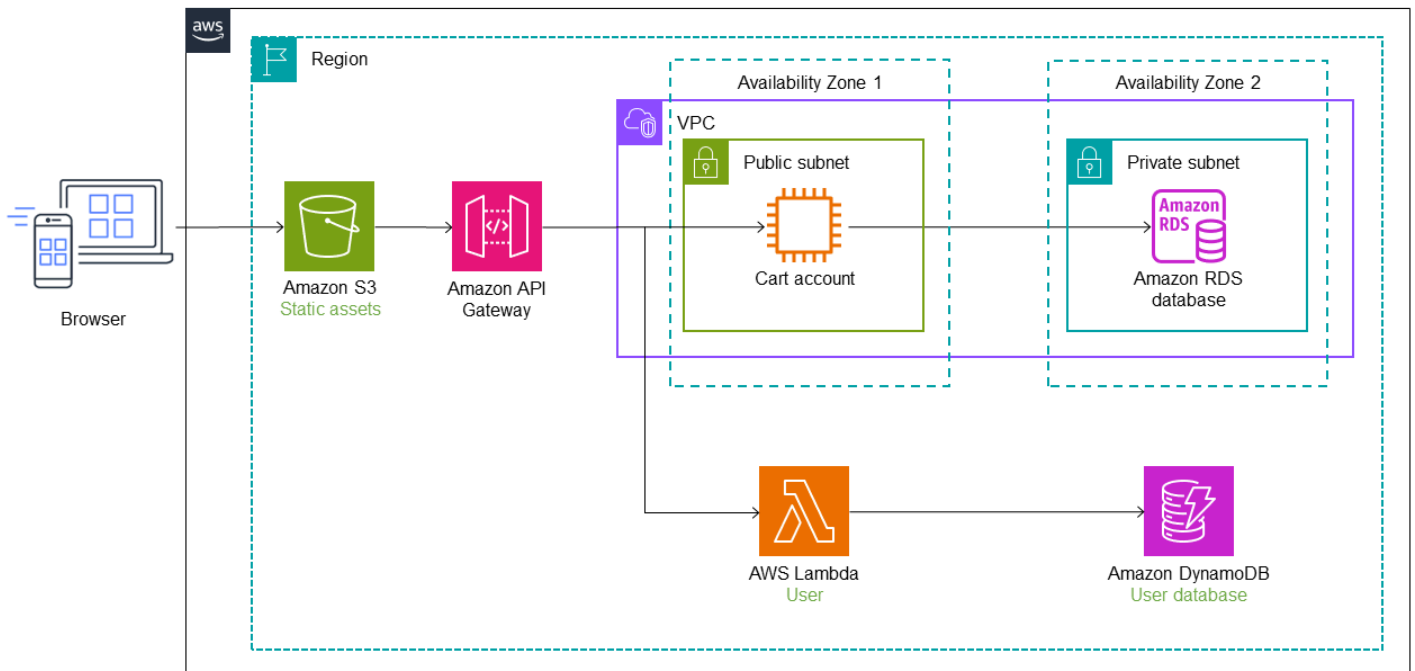
다음 다이어그램은 모놀리식 애플리케이션의 초기 상태를 보여줍니다. [Amazon Elastic Compute Cloud \(Amazon EC2\) 인스턴스에서 실행되고 Amazon RDS \(관계형 데이터베이스 서비스\) 데이터베이스를 사용하는 lift-and-shift 전략을 사용하여 마이그레이션했다고 AWS 가정에 보겠습니다.](#) 단순화를 위해 아키텍처는 하나의 사설과 하나의 퍼블릭 서브넷이 있는 단일 가상 사설 클라우드 (VPC) 를 사용합니다. 마이크로서비스가 처음에 동일한 서브넷 내에 배포된다고 가정해 보겠습니다. AWS 계정 (프로덕션 환경의 모범 사례는 다중 계정 아키텍처를 사용하여 배포의 독립성을 보장하는 것입니다.) EC2 인스턴스는 퍼블릭 서브넷의 단일 가용 영역에 상주하고, RDS 인스턴스는 프라이빗 서브넷의 단일 가용 영역에 있습니다. [Amazon Simple Storage Service \(Amazon S3\)](#) 는 웹 사이트의, CSS 및 React 파일과 같은 JavaScript 정적 자산을 저장합니다.



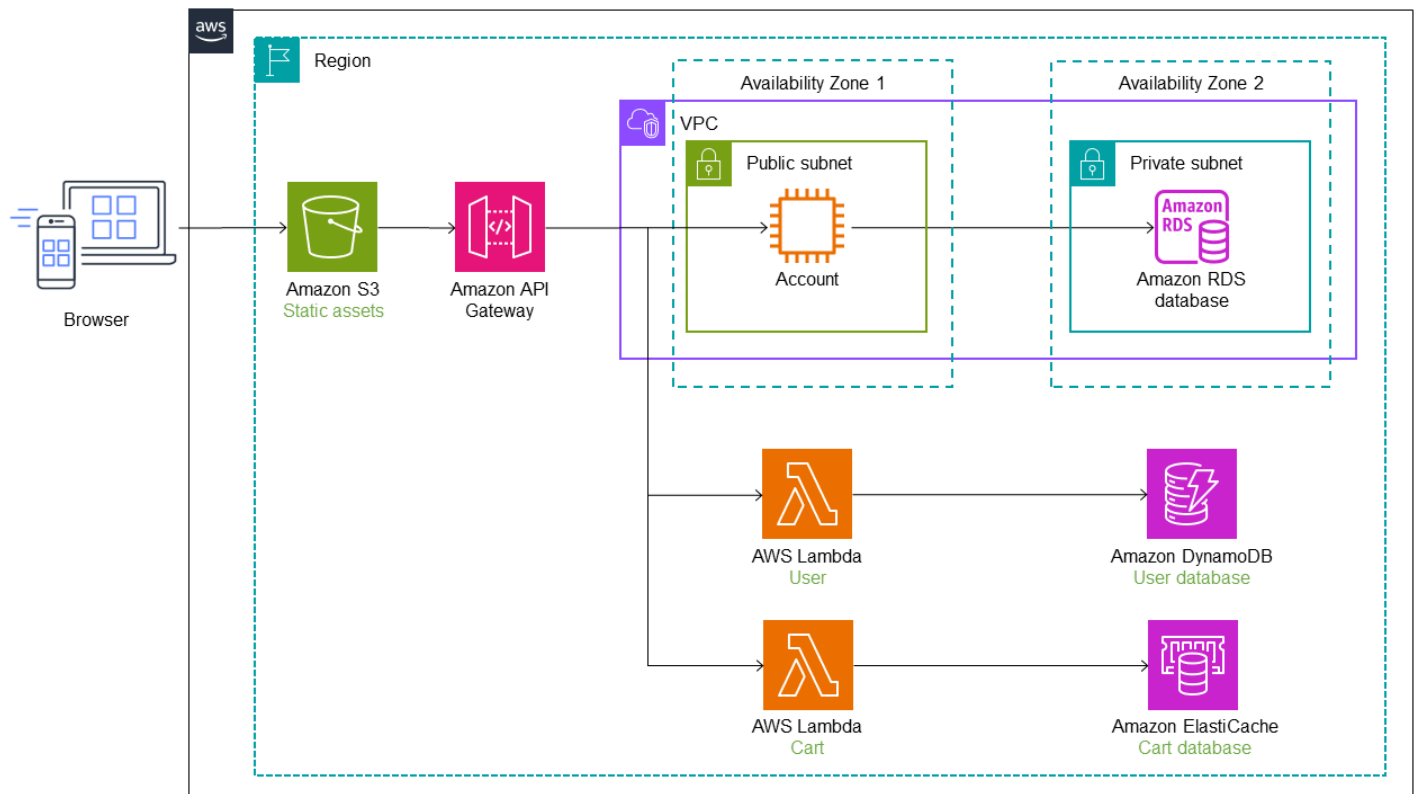
다음 아키텍처에서는 [Amazon API Gateway](#)를 모놀리식 애플리케이션 앞에 [AWS Migration Hub Refactor Spaces](#) 배포합니다. Refactor Spaces는 계정 내에 리팩토링 인프라를 생성하고 API Gateway는 모놀리식으로 호출을 라우팅하기 위한 프록시 계층 역할을 합니다. 처음에는 모든 호출이 프록시 레이어를 통해 모놀리식 애플리케이션으로 라우팅됩니다. 앞서 설명한 것처럼 프록시 계층은 단일 장애 지점이 될 수 있습니다. 하지만 API Gateway를 프록시로 사용하면 서버리스 다중 AZ 서비스이므로 위험을 완화할 수 있습니다.



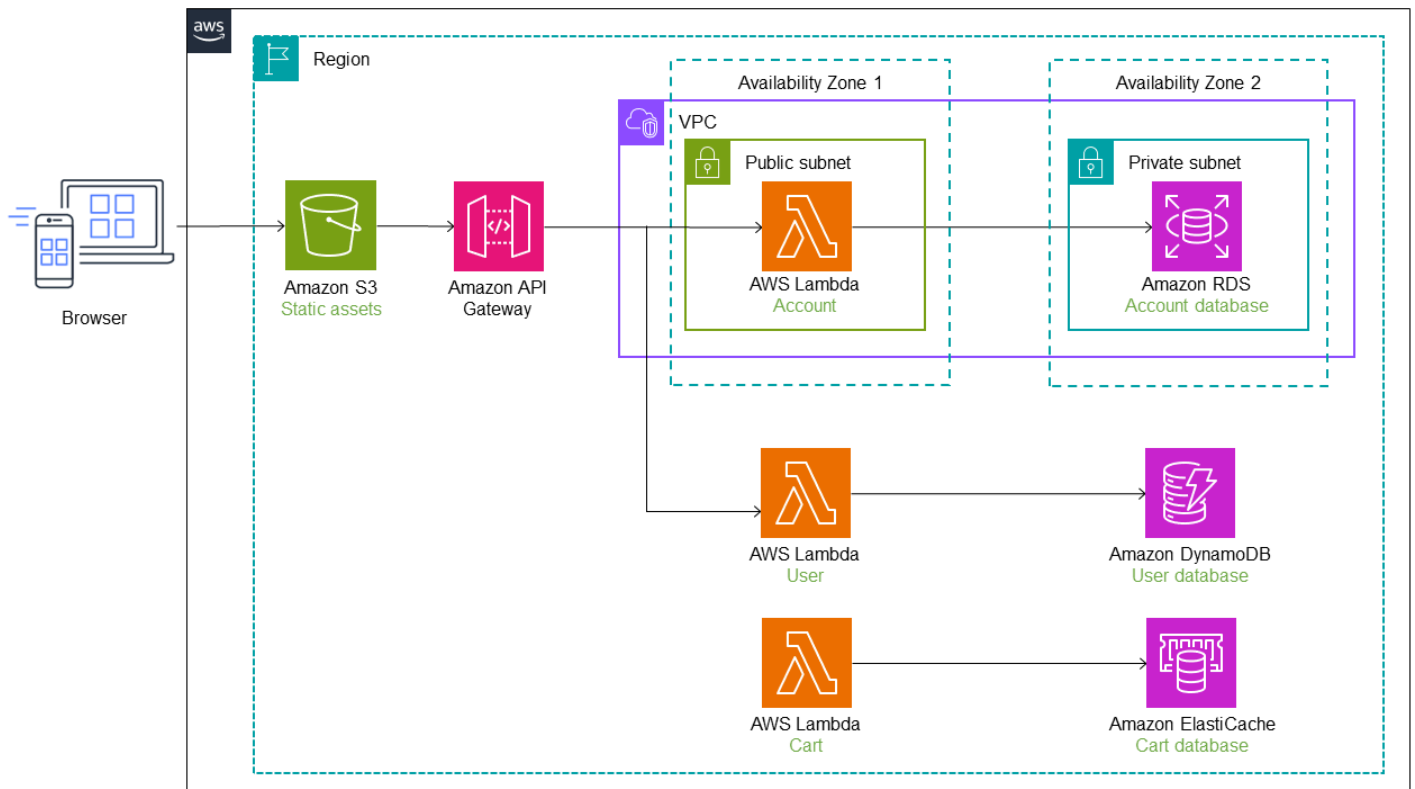
사용자 서비스가 Lambda 함수로 마이그레이션되고 Amazon [DynamoDB](#) 데이터베이스가 해당 데이터를 저장합니다. Lambda 서비스 엔드포인트와 기본 경로가 리팩터링 스페이스에 추가되고 API Gateway는 호출을 Lambda 함수로 라우팅하도록 자동으로 구성됩니다. [구현 세부 정보는 반복적 애플리케이션 현대화 워크숍의 모듈 2를 참조하십시오.](#)



다음 다이어그램에서 카트 서비스도 모놀리스에서 Lambda 함수로 마이그레이션되었습니다. 추가 경로 및 서비스 엔드포인트가 Refactor Spaces에 추가되고 트래픽이 자동으로 Cart Lambda 함수로 전환됩니다. [Lambda 함수의 데이터 스토어는 Amazon에서 관리합니다.](#) [ElastiCache](#) 모놀리식 애플리케이션은 여전히 Amazon RDS 데이터베이스와 함께 EC2 인스턴스에 남아 있습니다.



다음 다이어그램에서는 마지막 서비스 (계정) 가 모놀리스에서 Lambda 함수로 마이그레이션됩니다. 원래 Amazon RDS 데이터베이스를 계속 사용합니다. 이제 새 아키텍처에는 별도의 데이터베이스가 있는 세 개의 마이크로서비스가 있습니다. 각 서비스는 서로 다른 유형의 데이터베이스를 사용합니다. 마이크로서비스의 특정 요구 사항을 충족하기 위해 특별히 구축된 데이터베이스를 사용하는 이러한 개념을 다중 언어 지속성이라고 합니다. Lambda 함수는 사용 사례에 따라 다양한 프로그래밍 언어로 구현될 수도 있습니다. 리팩토링 중에 Refactor Spaces는 Lambda로의 트래픽 컷오버 및 라우팅을 자동화합니다. 이를 통해 빌더는 라우팅 인프라를 설계, 배포 및 구성하는 데 필요한 시간을 절약할 수 있습니다.

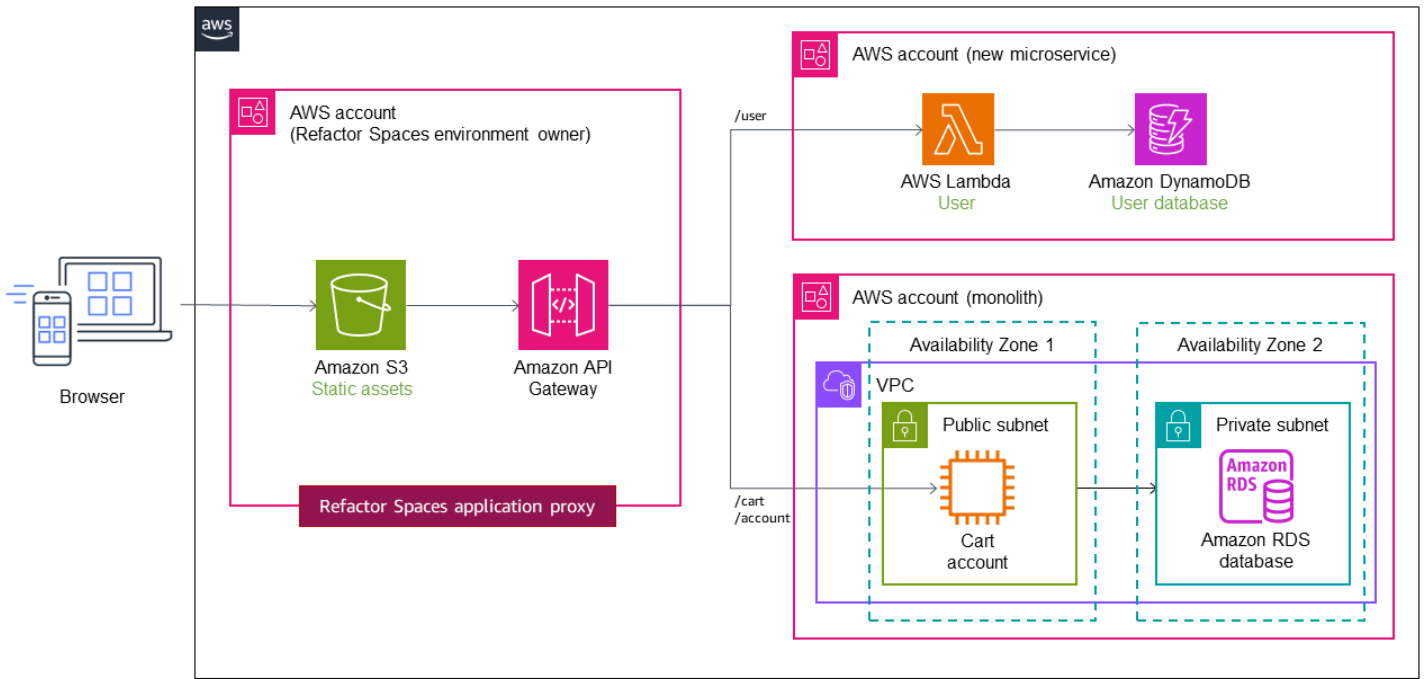


여러 계정 사용

이전 구현에서는 모놀리식 애플리케이션에 프라이빗 서브넷과 퍼블릭 서브넷이 있는 단일 VPC를 사용했으며, 단순성을 위해 동일한 VPC 내에 마이크로서비스를 배포했습니다. AWS 계정 하지만 마이크로서비스가 배포 독립성을 위해 여러 개로 배포되는 경우가 많은 실제 시나리오에서는 이런 경우가 거의 없습니다. AWS 계정 다중 계정 구조에서는 모놀리식에서 다른 계정의 새 서비스로의 라우팅 트래픽을 구성해야 합니다.

[Refactor Spaces](#)를 사용하면 모놀리식 애플리케이션 외부로 API 호출을 라우팅하기 위한 AWS 인프라를 만들고 구성할 수 있습니다. 리팩터링은 AWS 계정 내의 [API Gateway](#), [Network Load Balancer](#) 및 [IAM \(AWS Identity and Access Management 리소스 기반\)](#) 정책을 애플리케이션 리소스의 일부로 오케스트레이션합니다. 단일 AWS 계정 또는 여러 계정의 새 서비스를 외부 HTTP 엔드포인트에 투명하게 추가할 수 있습니다. 이러한 모든 리소스는 내부에서 오케스트레이션되며 배포 후 사용자 지정 AWS 계정 및 구성할 수 있습니다.

다음 다이어그램과 같이 사용자 및 카드 서비스가 서로 다른 두 계정에 배포되었다고 가정해 보겠습니다. 리팩터링 스페이스를 사용할 때는 서비스 엔드포인트와 라우트만 구성하면 됩니다. Refactor Spaces는 [API Gateway—Lambda 통합 및 Lambda](#) 리소스 정책 생성을 자동화하므로, 사용자는 모놀리식에서 서비스를 안전하게 리팩토링하는 데 집중할 수 있습니다.



스페이스 리팩터링 사용에 대한 동영상 튜토리얼은 [앱을 사용하여 점진적으로 리팩터링하기](#) 항목을 참조하십시오. AWS Migration Hub Refactor Spaces

워크숍

- [반복적 앱 현대화 워크숍](#)

블로그 레퍼런스

- [AWS Migration Hub Refactor Spaces](#)
- [심층 분석 AWS Migration Hub Refactor Spaces](#)
- [배포 파이프라인, 참조 아키텍처 및 참조 구현](#)

관련 콘텐츠

- [API 라우팅 패턴](#)
- [리팩터링 스페이스 문서](#)

트랜잭션 아웃박스 패턴

의도

트랜잭션 아웃박스 패턴은 단일 작업에 데이터베이스 쓰기 작업과 메시지 또는 이벤트 알림이 모두 포함된 경우에 분산 시스템에서 발생하는 이중 쓰기 작업 문제를 해결합니다. 이중 쓰기 작업은 애플리케이션이 서로 다른 두 시스템에 데이터를 쓸 때 발생합니다. 예를 들어 마이크로서비스가 데이터베이스에 데이터를 보관하고 메시지를 전송하여 다른 시스템에 알려야 하는 경우가 이에 해당합니다. 이러한 작업 중 하나에서 오류가 발생하면 데이터가 일치하지 않게 될 수 있습니다.

목적

데이터베이스 업데이트 후 마이크로서비스가 이벤트 알림을 보내는 경우 데이터 일관성과 신뢰성을 보장하기 위해 이 두 작업이 원자적으로 실행되어야 합니다.

- 데이터베이스 업데이트는 성공했지만 이벤트 알림이 실패할 경우 다운스트림 서비스는 변경 사항을 인식하지 못해 시스템이 일관되지 않은 상태가 될 수 있습니다.
- 데이터베이스 업데이트에 실패했지만 이벤트 알림이 전송되면 데이터가 손상되어 시스템의 신뢰성에 영향을 미칠 수 있습니다.

적용 가능성

다음과 같은 경우 트랜잭션 아웃박스 패턴을 사용합니다.

- 데이터베이스 업데이트에서 이벤트 알림이 시작되는 이벤트 기반 애플리케이션을 구축하고 있습니다.
- 두 서비스를 포함하는 작업에서 원자성을 보장하고자 합니다.
- [이벤트 소싱 패턴](#)을 구현하고자 합니다.

문제 및 고려 사항

- 중복 메시지: 이벤트 처리 서비스에서 중복된 메시지나 이벤트를 보낼 수 있으므로, 처리된 메시지를 추적하여 소비하는 서비스를 멍등 상태로 만드는 것이 좋습니다.
- 알림 순서: 서비스에서 데이터베이스를 업데이트하는 순서와 동일한 순서로 메시지나 이벤트를 전송합니다. 이는 이벤트 저장소를 사용하여 데이터 저장소를 point-in-time 복구할 수 있는 이벤트 소

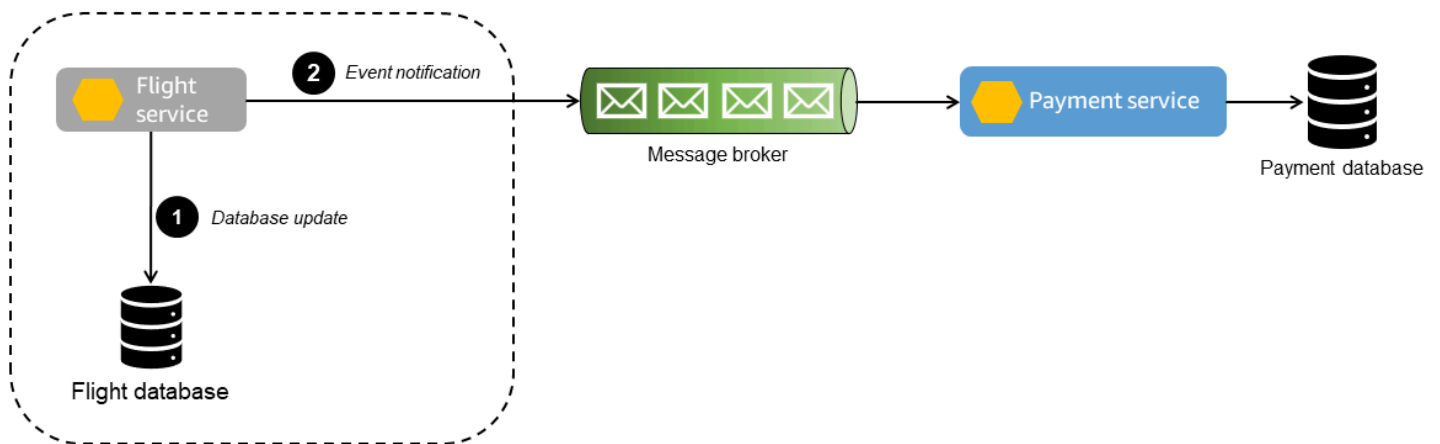
싱 패턴에 매우 중요합니다. 순서가 올바르지 않으면 데이터 품질이 저하될 수 있습니다. 알림 순서가 유지되지 않으면 최종 일관성과 데이터베이스 롤백으로 인해 문제가 복잡해질 수 있습니다.

- 트랜잭션 롤백: 트랜잭션이 롤백된 경우 이벤트 알림을 보내지 마세요.
- 서비스 수준 트랜잭션 처리: 여러 서비스에 걸친 트랜잭션에서 데이터 스토어 업데이트가 요구되는 경우, [Saga 오케스트레이션 패턴](#)을 사용하여 데이터 스토어 전체의 데이터 무결성을 유지합니다.

구현

전반적인 아키텍처

다음 시퀀스 다이어그램은 이중 쓰기 작업 중에 발생하는 이벤트의 순서를 보여줍니다.



1. 항공편 서비스가 데이터베이스에 데이터를 쓰고 결제 서비스에 이벤트 알림을 보냅니다.
2. 메시지 브로커가 메시지와 이벤트를 결제 서비스에 전달합니다. 메시지 브로커에 장애가 발생하면 결제 서비스가 업데이트를 수신할 수 없습니다.

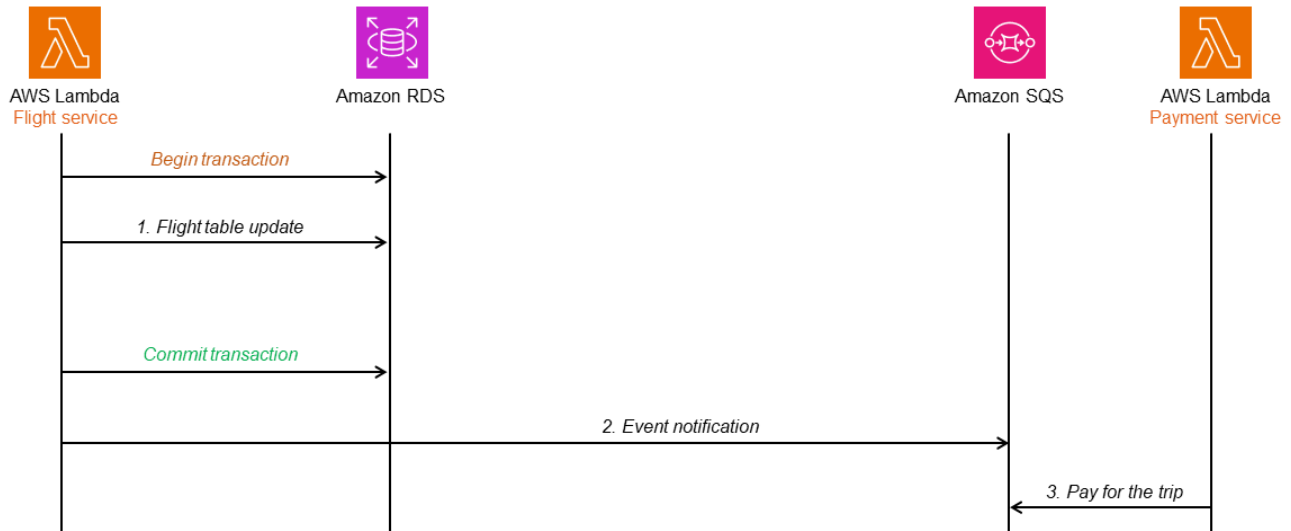
항공편 데이터베이스 업데이트에 실패했지만 알림은 전송될 경우 결제 서비스는 이벤트 알림에 따라 결제를 처리합니다. 이로 인해 다운스트림 데이터 불일치가 발생할 수 있습니다.

AWS 서비스를 사용한 구현

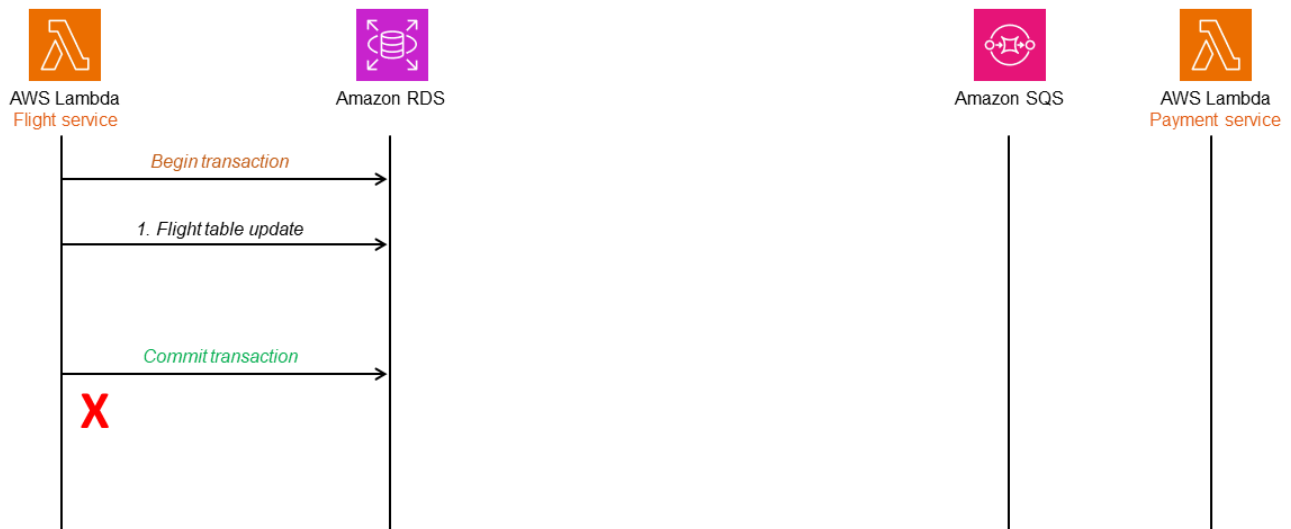
시퀀스 다이어그램에서 패턴을 설명하기 위해 다음 다이어그램에 표시된 대로 다음 AWS 서비스를 사용합니다.

- 마이크로서비스는 [AWS Lambda](#)를 사용하여 구현됩니다.
- 기본 데이터베이스는 [Amazon Relational Database Service\(RDS\)](#)에서 관리됩니다.

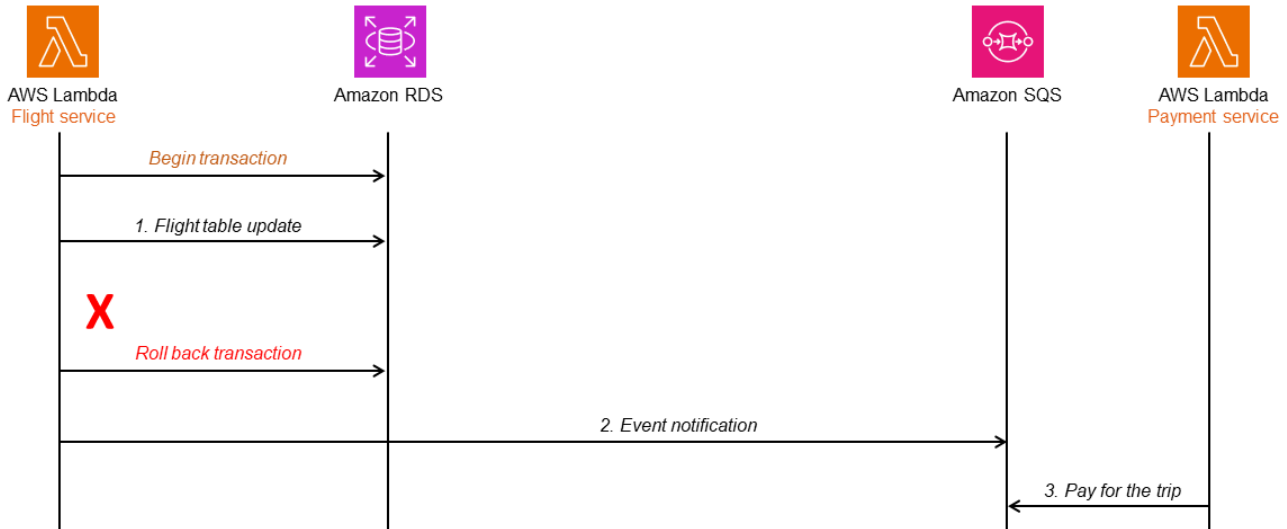
- [Amazon Simple Queue Service\(Amazon SQS\)](#)는 이벤트 알림을 수신하는 메시지 브로커 역할을 합니다.



거래를 커밋한 후 항공편 서비스에 장애가 발생하면 이벤트 알림이 전송되지 않을 수 있습니다.



하지만 트랜잭션이 실패하고 롤백되더라도 이벤트 알림이 계속 전송되어 결제 서비스가 결제를 처리하게 될 수 있습니다.



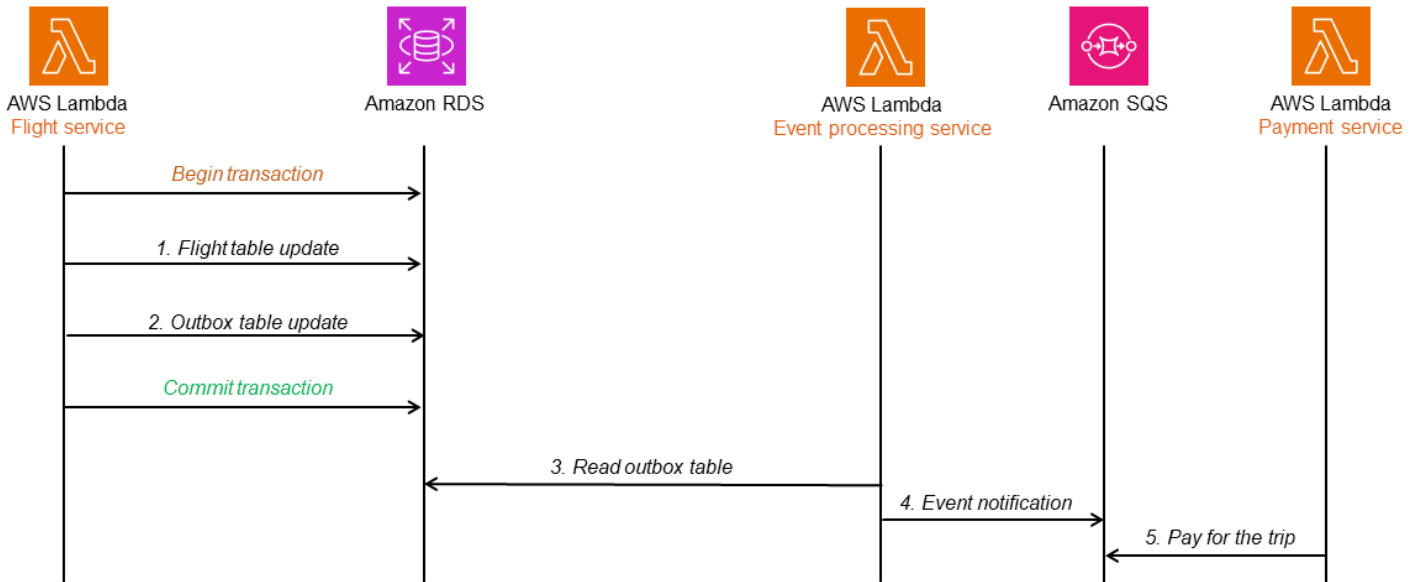
이 문제를 해결하려면 아웃박스 테이블을 사용하거나 변경 데이터 캡처(CDC) 기능을 사용할 수 있습니다. 다음 섹션에서는 이 두 가지 옵션과 AWS 서비스를 사용하여 이를 구현하는 방법을 설명합니다.

관계형 데이터베이스에 아웃박스 테이블 사용

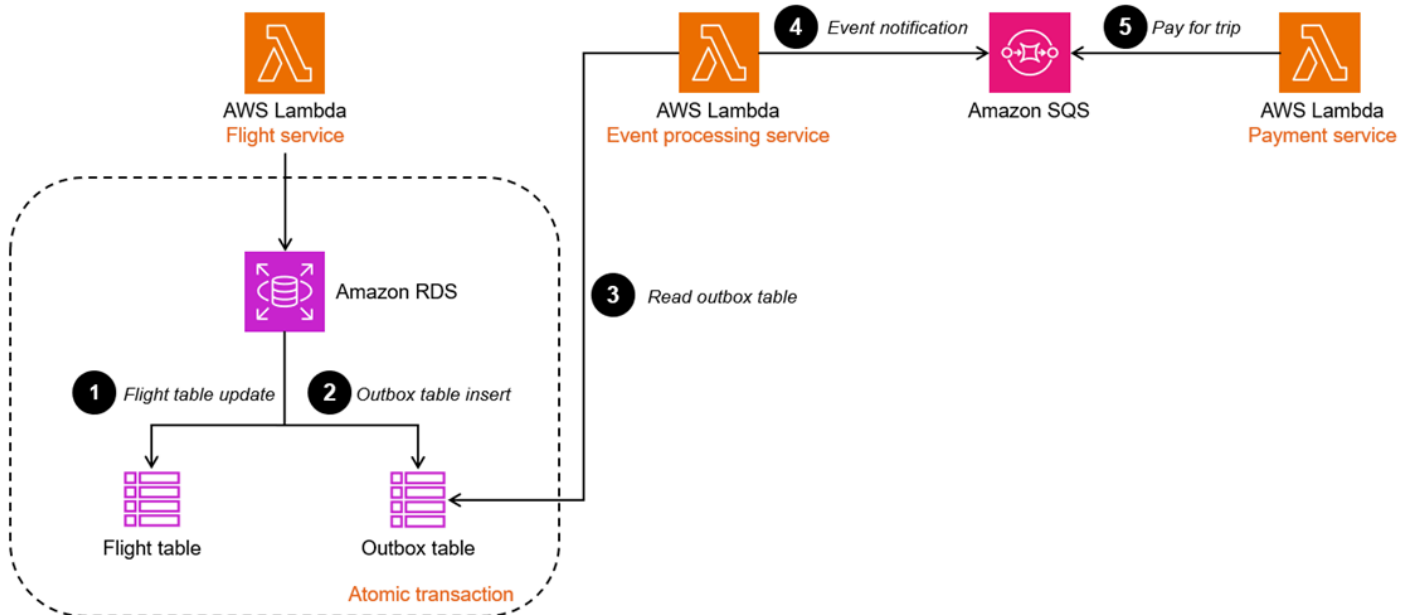
아웃박스 테이블에는 항공편 서비스의 모든 이벤트가 타임스탬프 및 시퀀스 번호와 함께 저장됩니다.

항공편 테이블이 업데이트되면 아웃박스 테이블도 동일한 트랜잭션에서 업데이트됩니다. 또 다른 서비스(예: 이벤트 처리 서비스)가 아웃박스 테이블에서 데이터를 읽고 Amazon SQS로 이벤트를 전송합니다. Amazon SQS는 추가 처리를 위해 이벤트 관련 메시지를 결제 서비스에 전송합니다. [Amazon SQS 표준 대기열](#)은 메시지가 한 번 이상 전송되고 손실되지 않도록 보장합니다. 하지만 Amazon SQS 표준 대기열을 사용하는 경우 동일한 메시지 또는 이벤트가 두 번 이상 전송될 수 있으므로, 이벤트 알림 서비스가 멍등적인지 확인해야 합니다(즉, 동일한 메시지를 여러 번 처리해도 부작용이 없어야 함). 메시지 순서 지정을 통해 메시지를 정확히 한 번만 전송해야 하는 경우 [Amazon SQS 선입선출\(FIFO\) 대기열](#)을 사용할 수 있습니다.

항공편 테이블 업데이트가 실패하거나 아웃박스 테이블 업데이트가 실패할 경우, 전체 트랜잭션을 롤백하므로 다운스트림 데이터 불일치가 발생하지 않습니다.



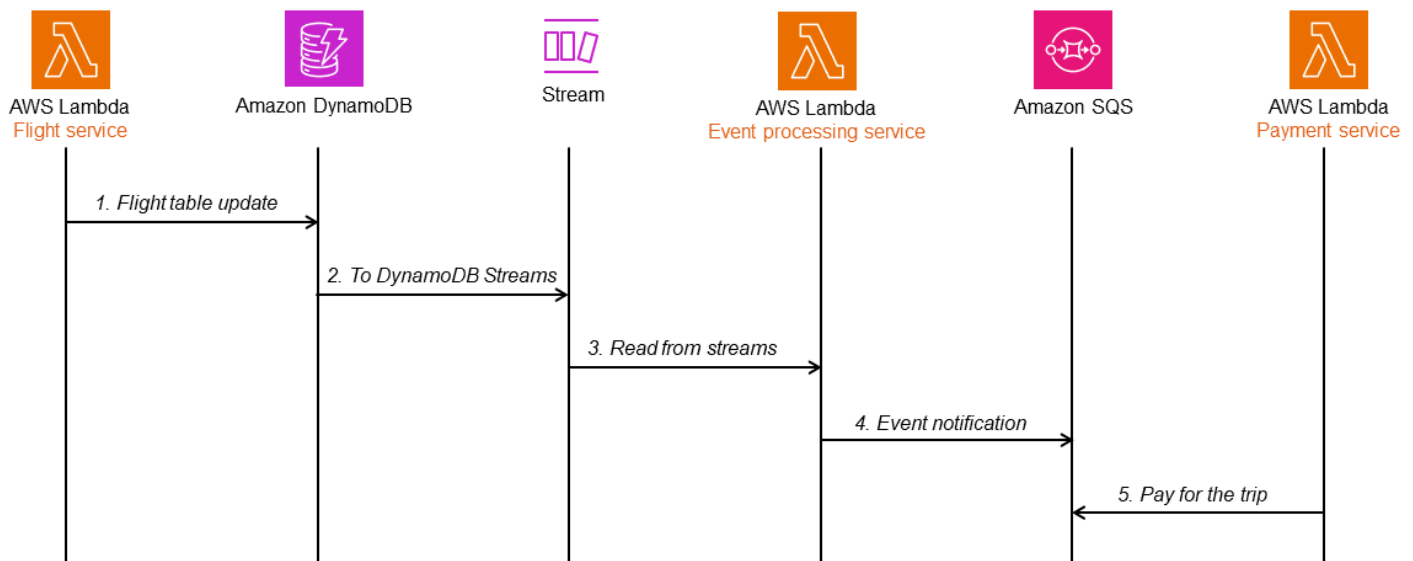
다음 다이어그램에서는 Amazon RDS 데이터베이스를 사용하여 트랜잭션 아웃박스 아키텍처를 구현했습니다. 이벤트 처리 서비스는 아웃박스 테이블을 읽을 때 커밋된 (성공적인) 트랜잭션에 포함된 행만 인식한 다음, 해당 이벤트에 대한 메시지를 SQS 대기열에 추가합니다. 결제 서비스가 추가 처리를 위해 이 대기열을 읽습니다. 이 설계 방식은 이중 쓰기 작업을 해결하고 타임스탬프와 시퀀스 번호를 사용하여 메시지와 이벤트의 순서를 유지합니다.



변경 데이터 캡처 (CDC) 사용

일부 데이터베이스는 변경된 데이터를 캡처하기 위한 항목 수준 수정 게시 기능을 지원합니다. 변경된 항목을 식별하고 그에 따라 이벤트 알림을 보낼 수 있습니다. 이렇게 하면 업데이트를 추적하기 위해 다른 테이블을 만드는 데 따른 오버헤드를 줄일 수 있습니다. 항공편 서비스가 시작한 이벤트는 동일한 항목의 다른 속성에 저장됩니다.

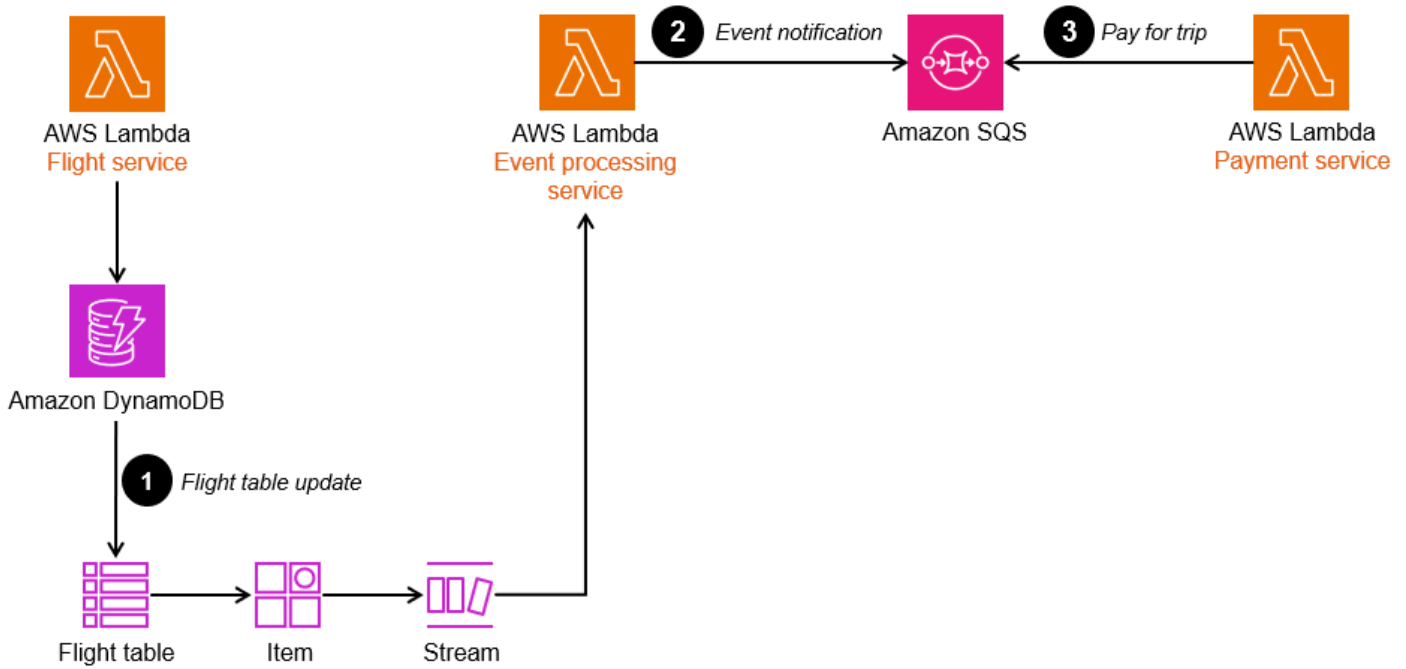
[Amazon DynamoDB](#)는 CDC 업데이트를 지원하는 키-값 NoSQL 데이터베이스입니다. 다음 시퀀스 다이어그램에서 DynamoDB는 Amazon DynamoDB Streams에 항목 수준 수정 사항을 게시합니다. 이벤트 처리 서비스는 스트림에서 데이터를 읽고 추가 처리를 위해 이벤트 알림을 결제 서비스에 게시합니다.



DynamoDB Streams는 시간 순서에 따라 DynamoDB 테이블의 항목 수준 변경 사항과 관련된 정보의 흐름을 캡처합니다.

DynamoDB 테이블에서 스트림을 활성화하여 트랜잭션 아웃박스 패턴을 구현할 수 있습니다. 이벤트 처리 서비스를 위한 Lambda 함수가 이러한 스트림과 연결됩니다.

- 항공편 테이블이 업데이트되면 DynamoDB Streams가 변경된 데이터를 캡처하고 이벤트 처리 서비스가 스트림을 폴링하여 새 레코드를 찾습니다.
- 새 스트림 레코드를 사용할 수 있게 되면 Lambda 함수는 추가 처리를 위해 이벤트에 대한 메시지를 동기식으로 SQS 대기열에 추가합니다. DynamoDB 항목에 속성을 추가하여 필요에 따라 타임스탬프와 시퀀스 번호를 캡처함으로써 구현의 안정성을 개선할 수 있습니다.



샘플 코드

아웃박스 테이블 사용

이 섹션의 샘플 코드는 아웃박스 테이블을 사용하여 트랜잭션 아웃박스 패턴을 구현하는 방법을 보여줍니다. 전체 코드를 보려면 이 예제의 [GitHub 리포지토리를](#) 참조하십시오.

다음 코드 조각은 데이터베이스의 Flight 엔터티와 Flight 이벤트를 단일 트랜잭션 내의 각 테이블에 저장합니다.

```

@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;
}
  
```

아웃박스 테이블에서 새 이벤트를 주기적으로 스캔하여 Amazon SQS로 전송하고, Amazon SQS가 성공적으로 응답하면 테이블에서 해당 이벤트를 삭제하는 작업은 별도의 서비스가 담당합니다. 폴링 속도는 `application.properties` 파일에서 구성할 수 있습니다.

```
@Scheduled(fixedDelayString = "${sqs.polling_ms}")
public void forwardEventsToSQS() {
    List<FlightOutbox> entities =
outboxRepository.findAllByOrderByAsc(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);
    }
}
```

변경 데이터 캡처 (CDC) 사용

이 섹션의 샘플 코드는 DynamoDB의 변경 데이터 캡처 (CDC) 기능을 사용하여 트랜잭션 아웃박스 패턴을 구현하는 방법을 보여줍니다. [전체 코드를 보려면 이 예제의 리포지토리를 참조하십시오. GitHub](#)

다음 AWS Cloud Development Kit (AWS CDK) 코드 스니펫은 DynamoDB 플라이트 테이블과 Amazon Kinesis 데이터 스트림 (cdcStream) 을 생성하고, 스트림에 모든 업데이트를 전송하도록 플라이트 테이블을 구성합니다.

```
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,
        }
    });

```

다음 코드 스니펫 및 구성은 Kinesis 스트림에서 업데이트를 선택하고 추가 처리를 위해 이러한 이벤트를 SQS 대기열로 전달하는 스프링 클라우드 스트림 함수를 정의합니다.

```

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 리포지토리

이 패턴의 샘플 아키텍처를 완전히 구현하려면 <https://github.com/aws-samples/GitHub> 리포지토리를 참조하십시오 `transactional-outbox-pattern`.

리소스

참조

- [AWS 아키텍처 센터](#)
- [AWS 개발자 센터](#)
- [아마존 빌더 라이브러리](#)

Tools

- [AWS Well-Architected Tool](#)
- [AWS 앱2 컨테이너](#)
- [AWS Microservice Extractor for .NET](#)

방법론

- [트웰브 팩터 앱](#)(애덤 위긴스의 ePub)
- 나이가드, 마이클 T. [풀어줘! : 프로덕션용 소프트웨어 설계 및 배포](#). 2판. 노스캐롤라이나주 롤리: 프래그매틱 북셀프, 2018.
- [다중 언어 지속성](#)(마틴 파울러의 블로그 포스트)
- [StranglerFigApplication](#)(마틴 파울러의 블로그 포스트)

문서 기록

아래 표에 이 가이드의 주요 변경 사항이 설명되어 있습니다. 향후 업데이트에 대한 알림을 받으려면 [RSS 피드](#)를 구독하십시오.

변경 사항	설명	날짜
새 패턴	육각형 아키텍처와 스캐터 게더링 이라는 두 가지 새로운 패턴을 추가했습니다.	2024년 5월 7일
새 코드 예시	변경 데이터 캡처 (CDC) 사용 사례의 샘플 코드를 트랜잭션 아웃박스 패턴 패턴에 추가했습니다.	2024년 2월 23일
새 코드 예시	<ul style="list-style-type: none"> 트랜잭션 아웃박스 패턴을 샘플 코드로 업데이트했습니다. Saga 코레오그래피와 Saga 오케스트레이션으로 대체된 오케스트레이션 및 코레오그래피 패턴에 대한 섹션을 제거했습니다. 	2023년 11월 16일
새 패턴	Saga 코레오그래피 , 구독-게시 , 이벤트 소싱 이라는 세 가지 새로운 패턴을 추가했습니다.	2023년 11월 14일
업데이트	스트랭글러 피그 패턴 구현 섹션을 업데이트했습니다.	2023년 10월 2일
최초 게시	이 첫 번째 릴리스에는 손상 방지 계층(ACL), API 라우팅, 회로 차단기, 오케스트레이션 및 코레오그래피, 백오프를 사용	2023년 7월 28일

한 재시도, Saga 오케스트레이션, 스트랭글러 피그, 트랜잭션 아웃박스 등 8가지 디자인 패턴이 포함되어 있습니다.

AWS 규범적 지침 용어집

다음은 규범적 지침에서 제공하는 AWS 전략, 가이드 및 패턴에서 일반적으로 사용되는 용어입니다. 용어집 항목을 제안하려면 용어집 끝에 있는 피드백 제공 링크를 사용하십시오.

숫자

7가지 전략

애플리케이션을 클라우드로 이전하기 위한 7가지 일반적인 마이그레이션 전략 이러한 전략은 Gartner가 2011년에 파악한 5가지 전략을 기반으로 하며 다음으로 구성됩니다.

- 리팩터링/리아키텍트 - 클라우드 네이티브 기능을 최대한 활용하여 애플리케이션을 이동하고 해당 아키텍처를 수정함으로써 민첩성, 성능 및 확장성을 개선합니다. 여기에는 일반적으로 운영 체제와 데이터베이스 이식이 포함됩니다. 예: 온프레미스 Oracle 데이터베이스를 Amazon Aurora PostgreSQL 호환 에디션으로 마이그레이션하십시오.
- 리플랫폼(리프트 앤드 리세이프) - 애플리케이션을 클라우드로 이동하고 일정 수준의 최적화를 도입하여 클라우드 기능을 활용합니다. 예: 온프레미스 Oracle 데이터베이스를 오라클용 Amazon RDS (Amazon RDS) 로 마이그레이션합니다. AWS 클라우드
- 재구매(드롭 앤드 슝) - 일반적으로 기존 라이선스에서 SaaS 모델로 전환하여 다른 제품으로 전환합니다. 예: 고객 관계 관리 (CRM) 시스템을 Salesforce.com으로 마이그레이션하십시오.
- 리호스팅(리프트 앤드 시프트) - 애플리케이션을 변경하지 않고 클라우드로 이동하여 클라우드 기능을 활용합니다. 예: 온프레미스 Oracle 데이터베이스를 EC2 인스턴스에서 Oracle로 마이그레이션합니다. AWS 클라우드
- 재배포(하이퍼바이저 수준의 리프트 앤 시프트) - 새 하드웨어를 구매하거나, 애플리케이션을 다시 작성하거나, 기존 운영을 수정하지 않고도 인프라를 클라우드로 이동합니다. 온프레미스 플랫폼에서 동일한 플랫폼의 클라우드 서비스로 서버를 마이그레이션합니다. 예: Microsoft Hyper-V 애플리케이션을 다음으로 마이그레이션하십시오. AWS
- 유지(보관) - 소스 환경에 애플리케이션을 유지합니다. 대규모 리팩터링이 필요하고 해당 작업을 나중에 연기하려는 애플리케이션과 비즈니스 차원에서 마이그레이션할 이유가 없어 유지하려는 레거시 애플리케이션이 여기에 포함될 수 있습니다.
- 사용 중지 - 소스 환경에서 더 이상 필요하지 않은 애플리케이션을 폐기하거나 제거합니다.

A

ABAC

[속성 기반 액세스](#) 제어를 참조하십시오.

추상화된 서비스

[관리형 서비스를](#) 참조하십시오.

산

[원자성, 일관성, 격리성, 내구성을](#) 참조하십시오.

능동-능동 마이그레이션

양방향 복제 도구 또는 이중 쓰기 작업을 사용하여 소스 데이터베이스와 대상 데이터베이스가 동기화된 상태로 유지되고, 두 데이터베이스 모두 마이그레이션 중 연결 애플리케이션의 트랜잭션을 처리하는 데이터베이스 마이그레이션 방법입니다. 이 방법은 일회성 전환이 필요한 대신 소규모의 제어된 배치로 마이그레이션을 지원합니다. [더 유연하지만 액티브-패시브 마이그레이션보다 더 많은 작업이 필요합니다.](#)

능동-수동 마이그레이션

소스 데이터베이스와 대상 데이터베이스가 동기화된 상태로 유지되지만 소스 데이터베이스만 연결 애플리케이션의 트랜잭션을 처리하고 데이터는 대상 데이터베이스로 복제되는 데이터베이스 마이그레이션 방법입니다. 대상 데이터베이스는 마이그레이션 중 어떤 트랜잭션도 허용하지 않습니다.

집계 함수

행 그룹에서 연산을 수행하고 그룹에 대한 단일 반환값을 계산하는 SQL 함수입니다. 집계 함수의 예로는 `MAX` 및 `SUM`이 있습니다.

AI

[인공 지능을](#) 참조하십시오.

AIOps

[인공 지능 운영을](#) 참조하십시오.

익명화

데이터세트에서 개인 정보를 영구적으로 삭제하는 프로세스입니다. 익명화는 개인 정보 보호에 도움이 될 수 있습니다. 익명화된 데이터는 더 이상 개인 데이터로 간주되지 않습니다.

안티 패턴

솔루션이 다른 솔루션보다 비생산적이거나 비효율적이거나 덜 효과적이어서 반복되는 문제에 자주 사용되는 솔루션입니다.

애플리케이션 제어

시스템을 멀웨어로부터 보호하기 위해 승인된 애플리케이션만 사용할 수 있는 보안 접근 방식입니다.

애플리케이션 포트폴리오

애플리케이션 구축 및 유지 관리 비용과 애플리케이션의 비즈니스 가치를 비롯하여 조직에서 사용하는 각 애플리케이션에 대한 세부 정보 모음입니다. 이 정보는 [포트폴리오 검색 및 분석 프로세스](#)의 핵심이며 마이그레이션, 현대화 및 최적화할 애플리케이션을 식별하고 우선순위를 정하는 데 도움이 됩니다.

인공 지능

컴퓨터 기술을 사용하여 학습, 문제 해결, 패턴 인식 등 일반적으로 인간과 관련된 인지 기능을 수행하는 것을 전문으로 하는 컴퓨터 과학 분야입니다. 자세한 내용은 [What is Artificial Intelligence?](#)를 참조하십시오.

인공 지능 운영(AIOps)

기계 학습 기법을 사용하여 운영 문제를 해결하고, 운영 인시던트 및 사용자 개입을 줄이고, 서비스 품질을 높이는 프로세스입니다. AWS 마이그레이션 전략에서 AIOps가 사용되는 방법에 대한 자세한 내용은 [운영 통합 가이드](#)를 참조하십시오.

비대칭 암호화

한 쌍의 키, 즉 암호화를 위한 퍼블릭 키와 복호화를 위한 프라이빗 키를 사용하는 암호화 알고리즘입니다. 퍼블릭 키는 복호화에 사용되지 않으므로 공유할 수 있지만 프라이빗 키에 대한 액세스는 엄격히 제한되어야 합니다.

원자성, 일관성, 격리성, 내구성(ACID)

오류, 정전 또는 기타 문제가 발생한 경우에도 데이터베이스의 데이터 유효성과 운영 신뢰성을 보장하는 소프트웨어 속성 세트입니다.

ABAC(속성 기반 액세스 제어)

부서, 직무, 팀 이름 등의 사용자 속성을 기반으로 세분화된 권한을 생성하는 방식입니다. 자세한 내용은 AWS Identity and Access Management (IAM) [설명서의 AWS ABAC](#) for를 참조하십시오.

신뢰할 수 있는 데이터 소스

가장 신뢰할 수 있는 정보 소스로 간주되는 기본 버전의 데이터를 저장하는 위치입니다. 익명화, 편집 또는 가명화와 같은 데이터 처리 또는 수정의 목적으로 신뢰할 수 있는 데이터 소스의 데이터를 다른 위치로 복사할 수 있습니다.

가용 영역

다른 가용 영역의 장애로부터 격리되고 동일한 지역 내 다른 가용 영역에 저렴하고 지연 시간이 짧은 네트워크 연결을 제공하는 별도의 위치. AWS 리전

AWS 클라우드 채택 프레임워크 (AWS CAF)

조직이 클라우드로 성공적으로 AWS 전환하기 위한 효율적이고 효과적인 계획을 개발하는 데 도움이 되는 지침 및 모범 사례 프레임워크입니다. AWS CAF는 지침을 관점이라고 하는 6가지 중점 영역, 즉 비즈니스, 사람, 거버넌스, 플랫폼, 보안, 운영으로 분류합니다. 비즈니스, 사람 및 거버넌스 관점은 비즈니스 기술과 프로세스에 초점을 맞추고, 플랫폼, 보안 및 운영 관점은 전문 기술과 프로세스에 중점을 둡니다. 예를 들어, 사람 관점은 인사(HR), 직원 배치 기능 및 인력 관리를 담당하는 이해관계자를 대상으로 합니다. 이러한 관점에서 AWS CAF는 조직이 성공적인 클라우드 채택을 준비할 수 있도록 인력 개발, 교육 및 커뮤니케이션에 대한 지침을 제공합니다. 자세한 내용은 [AWS CAF 웹 사이트](#)와 [AWS CAF 백서](#)를 참조하십시오.

AWS 워크로드 검증 프레임워크 (AWS WQF)

데이터베이스 마이그레이션 워크로드를 평가하고 마이그레이션 전략을 권장하며 작업 예상치를 제공하는 도구입니다. AWS WQF는 () 에 포함됩니다. AWS Schema Conversion Tool AWS SCT데이터베이스 스키마 및 코드 객체, 애플리케이션 코드, 종속성 및 성능 특성을 분석하고 평가 보고서를 제공합니다.

B

배드 봇

개인이나 조직을 방해하거나 피해를 입히려는 의도를 가진 [봇입니다](#).

BCP

[비즈니스 연속성 계획을](#) 참조하십시오.

동작 그래프

리소스 동작과 시간 경과에 따른 상호 작용에 대한 통합된 대화형 뷰입니다. Amazon Detective에서 동작 그래프를 사용하여 실패한 로그인 시도, 의심스러운 API 호출 및 유사한 작업을 검사할 수 있습니다. 자세한 내용은 Detective 설명서의 [Data in a behavior graph](#)를 참조하십시오.

빅 엔디안 시스템

가장 중요한 바이트를 먼저 저장하는 시스템입니다. [엔디안도](#) 참조하십시오.

바이너리 분류

바이너리 결과(가능한 두 클래스 중 하나)를 예측하는 프로세스입니다. 예를 들어, ML 모델이 “이 이메일이 스팸인가요, 스팸이 아닌가요?”, ‘이 제품은 책임가요, 자동차인가요?’ 등의 문제를 예측해야 할 수 있습니다.

블룸 필터

요소가 세트의 멤버인지 여부를 테스트하는 데 사용되는 메모리 효율성이 높은 확률론적 데이터 구조입니다.

블루/그린(Blue/Green) 배포

서로 다르지만 동일한 환경을 두 개 만드는 배포 전략입니다. 현재 애플리케이션 버전을 한 환경 (파란색) 에서 실행하고 다른 환경 (녹색) 에서 새 애플리케이션 버전을 실행합니다. 이 전략을 사용하면 영향을 최소화하면서 신속하게 롤백할 수 있습니다.

bot

인터넷을 통해 자동화된 작업을 실행하고 사람의 활동이나 상호 작용을 시뮬레이션하는 소프트웨어 애플리케이션입니다. 인터넷에서 정보를 인덱싱하는 웹 크롤러와 같은 일부 봇은 유용하거나 유용합니다. 배드 봇으로 알려진 일부 다른 봇은 개인이나 조직을 방해하거나 피해를 입히기 위한 것입니다.

봇넷

[멀웨어에 감염되어 봇 허더 또는 봇 운영자로 알려진 단일 당사자의 통제 하에 있는 봇 네트워크](#). 봇넷은 봇과 그 영향을 확장하는 가장 잘 알려진 메커니즘입니다.

브랜치

코드 리포지토리의 포함된 영역입니다. 리포지토리에 생성되는 첫 번째 브랜치가 기본 브랜치입니다. 기존 브랜치에서 새 브랜치를 생성한 다음 새 브랜치에서 기능을 개발하거나 버그를 수정할 수 있습니다. 기능을 구축하기 위해 생성하는 브랜치를 일반적으로 기능 브랜치라고 합니다. 기능을 출시할 준비가 되면 기능 브랜치를 기본 브랜치에 다시 병합합니다. 자세한 내용은 [브랜치 정보](#) (문서) 를 참조하십시오. GitHub

브레이크 글래스 액세스

예외적인 상황에서 승인된 프로세스를 통해 사용자가 일반적으로 액세스 권한이 없는 데이터에 빠르게 액세스할 수 있는 AWS 계정 있는 수단입니다. 자세한 내용은 Well-Architected AWS 지침의 [브레이크 글래스 절차 구현](#) 표시기를 참조하십시오.

브라운필드 전략

사용자 환경의 기존 인프라 시스템 아키텍처에 브라운필드 전략을 채택할 때는 현재 시스템 및 인프라의 제약 조건을 중심으로 아키텍처를 설계합니다. 기존 인프라를 확장하는 경우 브라운필드 전략과 [그린필드](#) 전략을 혼합할 수 있습니다.

버퍼 캐시

가장 자주 액세스하는 데이터가 저장되는 메모리 영역입니다.

사업 역량

기업이 가치를 창출하기 위해 하는 일(예: 영업, 고객 서비스 또는 마케팅)입니다. 마이크로서비스 아키텍처 및 개발 결정은 비즈니스 역량에 따라 이루어질 수 있습니다. 자세한 내용은 백서의 [AWS에서 컨테이너화된 마이크로서비스 실행의 비즈니스 역량 중심의 구성화](#) 섹션을 참조하십시오.

비즈니스 연속성 계획(BCP)

대규모 마이그레이션과 같은 중단 이벤트가 운영에 미치는 잠재적 영향을 해결하고 비즈니스가 신속하게 운영을 재개할 수 있도록 지원하는 계획입니다.

C

CAF

[클라우드 채택 프레임워크를 참조하십시오AWS](#).

카나리아 배포

최종 사용자에게 버전을 느리고 점진적으로 릴리스하는 것입니다. 확신이 들면 새 버전을 배포하고 현재 버전을 완전히 교체합니다.

CCoE

[클라우드 센터 오브 엑셀런스를 참조하십시오](#).

CDC

[변경 데이터 캡처를 참조하십시오](#).

변경 데이터 캡처(CDC)

데이터베이스 테이블과 같은 데이터 소스의 변경 내용을 추적하고 변경 사항에 대한 메타데이터를 기록하는 프로세스입니다. 대상 시스템의 변경 내용을 감사하거나 복제하여 동기화를 유지하는 등의 다양한 용도로 CDC를 사용할 수 있습니다.

카오스 엔지니어링

시스템의 복원력을 테스트하기 위해 의도적으로 장애나 장애를 일으키는 이벤트를 발생시키는 행위 [AWS Fault Injection Service \(AWS FIS\)](#) 를 사용하여 AWS 워크로드에 스트레스를 주는 실험을 수행하고 응답을 평가할 수 있습니다.

CI/CD

[지속적 통합 및 지속적 전달](#)을 참조하십시오.

분류

예측을 생성하는 데 도움이 되는 분류 프로세스입니다. 분류 문제에 대한 ML 모델은 이산 값을 예측합니다. 이산 값은 항상 서로 다릅니다. 예를 들어, 모델이 이미지에 자동차가 있는지 여부를 평가해야 할 수 있습니다.

클라이언트측 암호화

대상이 데이터를 AWS 서비스 수신하기 전에 데이터를 로컬로 암호화합니다.

클라우드 혁신 센터(CCoE)

클라우드 모범 사례 개발, 리소스 동원, 마이그레이션 타임라인 설정, 대규모 혁신을 통한 조직 선도 등 조직 전체에서 클라우드 채택 노력을 추진하는 다분야 팀입니다. 자세한 내용은 AWS 클라우드 기업 전략 [블로그의 CCoE 게시물을](#) 참조하십시오.

클라우드 컴퓨팅

원격 데이터 스토리지와 IoT 디바이스 관리에 일반적으로 사용되는 클라우드 기술 클라우드 컴퓨팅은 일반적으로 [엣지 컴퓨팅 기술과](#) 연결됩니다.

클라우드 운영 모델

IT 조직에서 하나 이상의 클라우드 환경을 구축, 성숙화 및 최적화하는 데 사용되는 운영 모델입니다. 자세한 내용은 [클라우드 운영 모델 구축](#)을 참조하십시오.

클라우드 채택 단계

조직이 마이그레이션할 때 일반적으로 거치는 4단계는 다음과 같습니다. AWS 클라우드

- 프로젝트 - 개념 증명 및 학습 목적으로 몇 가지 클라우드 관련 프로젝트 실행
- 기반 - 클라우드 채택 확장을 위한 기초 투자(예: 랜딩 존 생성, CCoE 정의, 운영 모델 구축)
- 마이그레이션 - 개별 애플리케이션 마이그레이션
- Re-invention - 제품 및 서비스 최적화와 클라우드 혁신

Stephen Orban은 기업 전략 블로그의 [클라우드 우선주의를 향한 여정 및 채택 단계에 대한 블로그 게시물](#)에서 이러한 단계를 정의했습니다. AWS 클라우드 [이들이 AWS 마이그레이션 전략과 어떤 관련이 있는지에 대한 자세한 내용은 마이그레이션 준비 가이드를 참조하십시오.](#)

CMDB

[구성 관리 데이터베이스](#)를 참조하십시오.

코드 리포지토리

소스 코드와 설명서, 샘플, 스크립트 등의 기타 자산이 버전 관리 프로세스를 통해 저장되고 업데이트되는 위치입니다. 일반 클라우드 리포지토리에는 또는 이 포함됩니다 GitHub . AWS CodeCommit코드의 각 버전을 브랜치라고 합니다. 마이크로서비스 구조에서 각 리포지토리는 단일 기능 전용입니다. 단일 CI/CD 파이프라인은 여러 리포지토리를 사용할 수 있습니다.

콜드 캐시

비어 있거나, 제대로 채워지지 않았거나, 오래되었거나 관련 없는 데이터를 포함하는 버퍼 캐시입니다. 주 메모리나 디스크에서 데이터베이스 인스턴스를 읽어야 하기 때문에 성능에 영향을 미치며, 이는 버퍼 캐시에서 읽는 것보다 느립니다.

콜드 데이터

거의 액세스되지 않고 일반적으로 과거 데이터인 데이터. 이런 종류의 데이터를 쿼리할 때는 일반적으로 느린 쿼리가 허용됩니다. 이 데이터를 성능이 낮고 비용이 저렴한 스토리지 계층 또는 클래스로 옮기면 비용을 절감할 수 있습니다.

컴퓨터 비전 (CV)

기계 학습을 사용하여 디지털 이미지 및 비디오와 같은 시각적 형식에서 정보를 분석하고 추출하는 [AI](#) 분야. 예를 들어 AWS Panorama 는 온프레미스 카메라 네트워크에 CV를 추가하는 디바이스를 제공하고, SageMaker Amazon은 CV용 이미지 처리 알고리즘을 제공합니다.

구성 드리프트

워크로드의 경우 구성이 예상 상태에서 변경됩니다. 이로 인해 워크로드가 규정을 준수하지 않게 될 수 있으며, 일반적으로 점진적이고 의도하지 않은 방식으로 진행됩니다.

구성 관리 데이터베이스(CMDB)

하드웨어 및 소프트웨어 구성 요소와 해당 구성을 포함하여 데이터베이스와 해당 IT 환경에 대한 정보를 저장하고 관리하는 리포지토리입니다. 일반적으로 마이그레이션의 포트폴리오 검색 및 분석 단계에서 CMDB의 데이터를 사용합니다.

규정 준수 팩

AWS Config 규정 준수 및 보안 검사를 사용자 지정하기 위해 조합할 수 있는 규칙 및 수정 조치 모음입니다. YAML 템플릿을 사용하여 한 AWS 계정 및 지역 또는 조직 전체에 단일 엔티티로 적합성 팩을 배포할 수 있습니다. 자세한 내용은 설명서의 [적합성 팩](#)을 참조하십시오. AWS Config

지속적 통합 및 지속적 전달(CI/CD)

소프트웨어 릴리스 프로세스의 소스, 빌드, 테스트, 스테이징 및 프로덕션 단계를 자동화하는 프로세스입니다. CI/CD는 일반적으로 파이프라인으로 설명됩니다. CI/CD를 통해 프로세스를 자동화하고, 생산성을 높이고, 코드 품질을 개선하고, 더 빠르게 제공할 수 있습니다. 자세한 내용은 [지속적 전달의 이점](#)을 참조하십시오. CD는 지속적 배포를 의미하기도 합니다. 자세한 내용은 [지속적 전달\(Continuous Delivery\)](#)과 [지속적인 개발](#)을 참조하십시오.

CV

[컴퓨터 비전을 참조하십시오.](#)

D

저장 데이터

스토리지에 있는 데이터와 같이 네트워크에 고정되어 있는 데이터입니다.

데이터 분류

중요도와 민감도를 기준으로 네트워크의 데이터를 식별하고 분류하는 프로세스입니다. 이 프로세스는 데이터에 대한 적절한 보호 및 보존 제어를 결정하는 데 도움이 되므로 사이버 보안 위험 관리 전략의 중요한 구성 요소입니다. 데이터 분류는 AWS Well-Architected 프레임워크의 보안 핵심 요소입니다. 자세한 내용은 [데이터 분류](#)를 참조하십시오.

데이터 드리프트

프로덕션 데이터와 ML 모델 학습에 사용된 데이터 간의 상당한 차이 또는 시간 경과에 따른 입력 데이터의 의미 있는 변화. 데이터 드리프트는 ML 모델 예측의 전반적인 품질, 정확성 및 공정성을 저하시킬 수 있습니다.

전송 중 데이터

네트워크를 통과하고 있는 데이터입니다. 네트워크 리소스 사이를 이동 중인 데이터를 예로 들 수 있습니다.

데이터 메시

중앙 집중식 관리 및 거버넌스와 함께 분산되고 분산된 데이터 소유권을 제공하는 아키텍처 프레임워크입니다.

데이터 최소화

꼭 필요한 데이터만 수집하고 처리하는 원칙입니다. 에서 데이터 최소화를 실천하면 개인 정보 보호 위험, 비용 및 분석에 따른 탄소 발자국을 줄일 AWS 클라우드 수 있습니다.

데이터 경계

신뢰할 수 있는 ID만 예상 네트워크에서 신뢰할 수 있는 리소스에 액세스하도록 하는 데 도움이 되는 AWS 환경 내 일련의 예방 가드레일입니다. 자세한 내용은 [데이터 경계 구축을 참조하십시오](#).

AWS

데이터 사전 처리

원시 데이터를 ML 모델이 쉽게 구문 분석할 수 있는 형식으로 변환하는 것입니다. 데이터를 사전 처리한다는 것은 특정 열이나 행을 제거하고 누락된 값, 일관성이 없는 값 또는 중복 값을 처리함을 의미할 수 있습니다.

데이터 출처

라이프사이클 전반에 걸쳐 데이터의 출처와 기록을 추적하는 프로세스(예: 데이터 생성, 전송, 저장 방법).

데이터 주체

데이터를 수집 및 처리하는 개인입니다.

데이터 웨어하우스

분석과 같은 비즈니스 인텔리전스를 지원하는 데이터 관리 시스템. 데이터 웨어하우스에는 일반적으로 대량의 과거 데이터가 포함되며 일반적으로 쿼리 및 분석에 사용됩니다.

데이터 정의 언어(DDL)

데이터베이스에서 테이블 및 객체의 구조를 만들거나 수정하기 위한 명령문 또는 명령입니다.

데이터베이스 조작 언어(DML)

데이터베이스에서 정보를 수정(삽입, 업데이트 및 삭제)하기 위한 명령문 또는 명령입니다.

DDL

[데이터베이스 정의 언어](#)를 참조하십시오.

딥 앙상블

예측을 위해 여러 딥 러닝 모델을 결합하는 것입니다. 딥 앙상블을 사용하여 더 정확한 예측을 얻거나 예측의 불확실성을 추정할 수 있습니다.

딥 러닝

여러 계층의 인공 신경망을 사용하여 입력 데이터와 관심 대상 변수 간의 매핑을 식별하는 ML 하위 분야입니다.

defense-in-depth

네트워크와 그 안의 데이터 기밀성, 무결성 및 가용성을 보호하기 위해 컴퓨터 네트워크 전체에 일련의 보안 메커니즘과 제어를 신중하게 계층화하는 정보 보안 접근 방식입니다. 이 전략을 채택하면 AWS Organizations 구조의 여러 계층에 AWS 여러 컨트롤을 추가하여 리소스를 보호하는 데 도움이 됩니다. 예를 들어 다단계 인증, 네트워크 세분화, 암호화를 결합한 defense-in-depth 접근 방식을 사용할 수 있습니다.

위임된 관리자

에서 AWS Organizations 호환 가능한 서비스는 AWS 구성원 계정을 등록하여 조직의 계정을 관리하고 해당 서비스에 대한 권한을 관리할 수 있습니다. 이러한 계정을 해당 서비스의 위임된 관리자라고 합니다. 자세한 내용과 호환되는 서비스 목록은 AWS Organizations 설명서의 [AWS Organizations와 함께 사용할 수 있는 AWS 서비스](#)를 참조하십시오.

배포

대상 환경에서 애플리케이션, 새 기능 또는 코드 수정 사항을 사용할 수 있도록 하는 프로세스입니다. 배포에는 코드 베이스의 변경 사항을 구현한 다음 애플리케이션 환경에서 해당 코드베이스를 구축하고 실행하는 작업이 포함됩니다.

개발 환경

[환경](#)을 참조하십시오.

탐지 제어

이벤트 발생 후 탐지, 기록 및 알림을 수행하도록 설계된 보안 제어입니다. 이러한 제어는 기존의 예방적 제어를 우회한 보안 이벤트를 알리는 2차 방어선입니다. 자세한 내용은 Implementing security controls on AWS의 [Detective controls](#)를 참조하십시오.

개발 가치 흐름 매핑 (DVSM)

소프트웨어 개발 라이프사이클에서 속도와 품질에 부정적인 영향을 미치는 제약 조건을 식별하고 우선 순위를 지정하는 데 사용되는 프로세스입니다. DVSM은 원래 린 제조 방식을 위해 설계된 가치 흐름 매핑 프로세스를 확장합니다. 소프트웨어 개발 프로세스를 통해 가치를 창출하고 이동하는 데 필요한 단계와 팀에 중점을 둡니다.

디지털 트윈

건물, 공장, 산업 장비 또는 생산 라인과 같은 실제 시스템을 가상으로 표현한 것입니다. 디지털 트윈은 예측 유지 보수, 원격 모니터링, 생산 최적화를 지원합니다.

치수 표

[스타 스키마에서](#) 팩트 테이블의 양적 데이터에 대한 데이터 속성을 포함하는 작은 테이블입니다. 차원 테이블 속성은 일반적으로 텍스트처럼 동작하는 텍스트 필드 또는 불연속형 숫자입니다. 이러한 속성은 일반적으로 쿼리 제한, 필터링 및 결과 집합 레이블 지정에 사용됩니다.

재해

워크로드 또는 시스템이 기본 배포 위치에서 비즈니스 목표를 달성하지 못하게 방해하는 이벤트입니다. 이러한 이벤트는 자연재해, 기술적 오류, 의도하지 않은 구성 오류 또는 멀웨어 공격과 같은 사람의 행동으로 인한 결과일 수 있습니다.

재해 복구(DR)

[재해로 인한 다운타임과 데이터 손실을 최소화하기 위해 사용하는 전략과 프로세스입니다.](#) 자세한 내용은 [워크로드의 재해 복구 AWS: AWS Well-Architected 프레임워크에서의 클라우드 복구를 참조하십시오.](#)

DML

[데이터베이스](#) 조작 언어를 참조하십시오.

도메인 기반 설계

구성 요소를 각 구성 요소가 제공하는 진화하는 도메인 또는 핵심 비즈니스 목표에 연결하여 복잡한 소프트웨어 시스템을 개발하는 접근 방식입니다. 이 개념은 에릭 에반스에 의해 그의 저서인 도메인 기반 디자인: 소프트웨어 중심의 복잡성 해결(Boston: Addison-Wesley Professional, 2003)에서 소개되었습니다. Strangler Fig 패턴과 함께 도메인 기반 설계를 사용하는 방법에 대한 자세한 내용은 [컨테이너 및 Amazon API Gateway를 사용하여 기존의 Microsoft ASP.NET\(ASMX\) 웹 서비스를 점진적으로 현대화하는 방법](#)을 참조하십시오.

DR

[재해 복구](#)를 참조하십시오.

드리프트 감지

기존 구성으로부터의 편차 추적. 예를 들어 [시스템 리소스의 편차를 감지하는 AWS CloudFormation](#) 데 사용하거나 거버넌스 요구 사항 준수에 영향을 미칠 수 있는 [착륙 지대의 변경 사항을 탐지하는 AWS Control Tower](#) 데 사용할 수 있습니다.

DVSM

[개발 가치 흐름 매핑](#) 참조하십시오.

E

EDA

[탐색적 데이터 분석](#) 참조하십시오.

엣지 컴퓨팅

IoT 네트워크의 엣지에서 스마트 디바이스의 컴퓨팅 성능을 개선하는 기술 [클라우드 컴퓨팅과](#) 비교할 때 엣지 컴퓨팅은 통신 대기 시간을 줄이고 응답 시간을 개선할 수 있습니다.

암호화

사람이 읽을 수 있는 일반 텍스트 데이터를 암호문으로 변환하는 컴퓨팅 프로세스입니다.

암호화 키

암호화 알고리즘에 의해 생성되는 무작위 비트의 암호화 문자열입니다. 키의 길이는 다양할 수 있으며 각 키는 예측할 수 없고 고유하게 설계되었습니다.

엔디안

컴퓨터 메모리에 바이트가 저장되는 순서입니다. 빅 엔디안 시스템은 가장 중요한 바이트를 먼저 저장합니다. 리틀 엔디안 시스템은 가장 덜 중요한 바이트를 먼저 저장합니다.

엔드포인트

[서비스](#) 엔드포인트를 참조하십시오.

엔드포인트 서비스

Virtual Private Cloud(VPC)에서 호스팅하여 다른 사용자와 공유할 수 있는 서비스입니다. 다른 주체 AWS 계정 또는 AWS Identity and Access Management (IAM) 보안 주체에 권한을 부여하여 엔드포인트 서비스를 생성하고 권한을 부여할 수 있습니다. AWS PrivateLink 이러한 계정 또는 보안 주체는 인터페이스 VPC 엔드포인트를 생성하여 엔드포인트 서비스에 비공개로 연결할 수 있습니다.

다. 자세한 내용은 Amazon Virtual Private Cloud(VPC) 설명서의 [엔드포인트 서비스 생성](#)을 참조하십시오.

ERP (전사적 자원 관리)

기업의 주요 비즈니스 프로세스 (예: 회계, [MES](#), 프로젝트 관리) 를 자동화하고 관리하는 시스템입니다.

봉투 암호화

암호화 키를 다른 암호화 키로 암호화하는 프로세스입니다. 자세한 내용은 AWS Key Management Service (AWS KMS) [설명서의 봉투 암호화](#)를 참조하십시오.

환경

실행 중인 애플리케이션의 인스턴스입니다. 다음은 클라우드 컴퓨팅의 일반적인 환경 유형입니다.

- 개발 환경 - 애플리케이션 유지 관리를 담당하는 핵심 팀만 사용할 수 있는 실행 중인 애플리케이션의 인스턴스입니다. 개발 환경은 변경 사항을 상위 환경으로 승격하기 전에 테스트하는 데 사용됩니다. 이러한 유형의 환경을 테스트 환경이라고도 합니다.
- 하위 환경 - 초기 빌드 및 테스트에 사용되는 환경을 비롯한 애플리케이션의 모든 개발 환경입니다.
- 프로덕션 환경 - 최종 사용자가 액세스할 수 있는 실행 중인 애플리케이션의 인스턴스입니다. CI/CD 파이프라인에서 프로덕션 환경이 마지막 배포 환경입니다.
- 상위 환경 - 핵심 개발 팀 이외의 사용자가 액세스할 수 있는 모든 환경입니다. 프로덕션 환경, 프로덕션 이전 환경 및 사용자 수용 테스트를 위한 환경이 여기에 포함될 수 있습니다.

에픽

애자일 방법론에서 작업을 구성하고 우선순위를 정하는 데 도움이 되는 기능적 범주입니다. 에픽은 요구 사항 및 구현 작업에 대한 개괄적인 설명을 제공합니다. 예를 들어 AWS CAF 보안 에픽에는 ID 및 액세스 관리, 탐지 제어, 인프라 보안, 데이터 보호, 사고 대응 등이 포함됩니다. AWS 마이그레이션 전략의 에픽에 대한 자세한 내용은 [프로그램 구현 가이드](#)를 참조하십시오.

ERP

[엔터프라이즈 리소스 계획을](#) 참조하십시오.

탐색 데이터 분석(EDA)

데이터 세트를 분석하여 주요 특성을 파악하는 프로세스입니다. 데이터를 수집 또는 집계한 다음 초기 조사를 수행하여 패턴을 찾고, 이상을 탐지하고, 가정을 확인합니다. EDA는 요약 통계를 계산하고 데이터 시각화를 생성하여 수행됩니다.

F

팩트 테이블

[스타 스키마의](#) 중앙 테이블. 비즈니스 운영에 대한 정량적 데이터를 저장합니다. 일반적으로 팩트 테이블에는 측정값이 포함된 열과 차원 테이블의 외부 키가 포함된 열 등 두 가지 유형의 열이 포함됩니다.

빨리 실패하세요

빈번하고 점진적인 테스트를 통해 개발 라이프사이클을 단축하는 철학. 이는 애자일 접근 방식의 중요한 부분입니다.

장애 격리 경계

장애 영향을 제한하고 워크로드의 복원력을 개선하는 데 도움이 되는 가용 영역 AWS 리전, 컨트를 플레인 또는 데이터 플레인과 같은 경계 AWS 클라우드자세한 내용은 [AWS 장애 격리](#) 경계를 참조하십시오.

기능 브랜치

[브랜치를](#) 참조하십시오.

기능

예측에 사용하는 입력 데이터입니다. 예를 들어, 제조 환경에서 기능은 제조 라인에서 주기적으로 캡처되는 이미지일 수 있습니다.

기능 중요도

모델의 예측에 특성이 얼마나 중요한지를 나타냅니다. 이는 일반적으로 SHAP(Shapley Additive Descriptions) 및 통합 그래디언트와 같은 다양한 기법을 통해 계산할 수 있는 수치 점수로 표현됩니다. 자세한 내용은 [다음은AWS사용한 기계 학습 모델 해석 가능성을](#) 참조하십시오.

기능 변환

추가 소스로 데이터를 보강하거나, 값을 조정하거나, 단일 데이터 필드에서 여러 정보 세트를 추출하는 등 ML 프로세스를 위해 데이터를 최적화하는 것입니다. 이를 통해 ML 모델이 데이터를 활용할 수 있습니다. 예를 들어, 날짜 '2021-05-27 00:15:37'을 '2021년', '5월', '목', '15일'로 분류하면 학습 알고리즘이 다양한 데이터 구성 요소와 관련된 미묘한 패턴을 학습하는 데 도움이 됩니다.

FGAC

[세분화된 액세스 제어](#)를 참조하십시오.

세분화된 액세스 제어(FGAC)

여러 조건을 사용하여 액세스 요청을 허용하거나 거부합니다.

플래시컷 마이그레이션

단계별 접근 방식 대신 [변경 데이터 캡처를 통한 지속적인 데이터](#) 복제를 통해 최단 시간에 데이터를 마이그레이션하는 데이터베이스 마이그레이션 방법입니다. 목표는 가동 중지 시간을 최소화하는 것입니다.

G

지리적 차단

[지리적 제한](#)을 참조하십시오.

지리적 제한(지리적 차단)

CloudFrontAmazon에서는 특정 국가의 사용자가 콘텐츠 배포에 액세스하지 못하도록 하는 옵션을 제공합니다. 허용 목록 또는 차단 목록을 사용하여 승인된 국가와 차단된 국가를 지정할 수 있습니다. 자세한 내용은 [설명서의 콘텐츠의 지리적 배포 제한](#)을 참조하십시오. CloudFront

Gitflow 워크플로

하위 환경과 상위 환경이 소스 코드 리포지토리의 서로 다른 브랜치를 사용하는 방식입니다. Gitflow 워크플로는 레거시로 간주되며 [트렁크 기반 워크플로는](#) 현대적이고 선호되는 접근 방식입니다.

브라운필드 전략

새로운 환경에서 기존 인프라의 부재 시스템 아키텍처에 대한 그린필드 전략을 채택할 때 [브라운필드](#)라고도 하는 기존 인프라와의 호환성 제한 없이 모든 새로운 기술을 선택할 수 있습니다. 기존 인프라를 확장하는 경우 브라운필드 전략과 그린필드 전략을 혼합할 수 있습니다.

가드레일

조직 단위(OU) 전체에서 리소스, 정책 및 규정 준수를 관리하는 데 도움이 되는 중요 규칙입니다. 예방 가드레일은 규정 준수 표준에 부합하도록 정책을 시행하며, 서비스 제어 정책과 IAM 권한 경계를 사용하여 구현됩니다. 탐지 가드레일은 정책 위반 및 규정 준수 문제를 감지하고 해결을 위한 알림을 생성하며, 이들은, Amazon AWS Config AWS Security Hub GuardDuty AWS Trusted Advisor, Amazon Inspector 및 사용자 지정 AWS Lambda 검사를 사용하여 구현됩니다.

H

하

[고가용성을](#) 확인하세요.

이기종 데이터베이스 마이그레이션

다른 데이터베이스 엔진을 사용하는 대상 데이터베이스로 소스 데이터베이스 마이그레이션(예: Oracle에서 Amazon Aurora로) 이기종 마이그레이션은 일반적으로 리아키텍트 작업의 일부이며 스키마를 변환하는 것은 복잡한 작업일 수 있습니다. AWS 는 스키마 변환에 도움이 되는 [AWS SCT](#)를 제공합니다.

높은 가용성(HA)

문제나 재해 발생 시 개입 없이 지속적으로 운영할 수 있는 워크로드의 능력. HA 시스템은 자동으로 장애 조치되고, 지속적으로 고품질 성능을 제공하고, 성능에 미치는 영향을 최소화하면서 다양한 부하와 장애를 처리하도록 설계되었습니다.

히스토리언 현대화

제조 산업의 요구 사항을 더 잘 충족하도록 운영 기술(OT) 시스템을 현대화하고 업그레이드하는 데 사용되는 접근 방식입니다. 히스토리언은 공장의 다양한 출처에서 데이터를 수집하고 저장하는 데 사용되는 일종의 데이터베이스입니다.

동종 데이터베이스 마이그레이션

동일한 데이터베이스 엔진을 공유하는 대상 데이터베이스로 소스 데이터베이스 마이그레이션(예: Microsoft SQL Server에서 Amazon RDS for SQL Server로) 동종 마이그레이션은 일반적으로 리호스팅 또는 리플랫폼 작업의 일부입니다. 네이티브 데이터베이스 유틸리티를 사용하여 스키마를 마이그레이션할 수 있습니다.

핫 데이터

자주 액세스하는 데이터(예: 실시간 데이터 또는 최근 번역 데이터). 일반적으로 이 데이터에는 빠른 쿼리 응답을 제공하기 위한 고성능 스토리지 계층 또는 클래스가 필요합니다.

핫픽스

프로덕션 환경의 중요한 문제를 해결하기 위한 긴급 수정입니다. 긴급성 때문에 핫픽스는 일반적으로 일반적인 DevOps 릴리스 워크플로 외부에서 만들어집니다.

하이퍼케어 기간

전환 직후 마이그레이션 팀이 문제를 해결하기 위해 클라우드에서 마이그레이션된 애플리케이션을 관리하고 모니터링하는 기간입니다. 일반적으로 이 기간은 1~4일입니다. 하이퍼케어 기간이 끝나면 마이그레이션 팀은 일반적으로 애플리케이션에 대한 책임을 클라우드 운영 팀에 넘깁니다.

I

IaC

[인프라를 코드로 보세요.](#)

자격 증명 기반 정책

환경 내에서 권한을 정의하는 하나 이상의 IAM 보안 주체에 연결된 정책입니다. AWS 클라우드 유휴 애플리케이션

90일 동안 평균 CPU 및 메모리 사용량이 5~20%인 애플리케이션입니다. 마이그레이션 프로젝트에서는 이러한 애플리케이션을 사용 중지하거나 온프레미스에 유지하는 것이 일반적입니다.

IIoT

[산업용 사물 인터넷을 참조하십시오.](#)

불변의 인프라

기존 인프라를 업데이트, 패치 또는 수정하는 대신 프로덕션 워크로드용 새 인프라를 배포하는 모델입니다. [변경 불가능한 인프라는 기본적으로 변경 가능한 인프라보다 더 일관되고 안정적이며 예측 가능합니다.](#) 자세한 내용은 Well-Architected AWS 프레임워크의 [변경 불가능한 인프라를 사용한 배포](#) 모범 사례를 참조하십시오.

인바운드(수신) VPC

AWS 다중 계정 아키텍처에서 VPC는 애플리케이션 외부에서 네트워크 연결을 허용, 검사 및 라우팅합니다. [AWS Security Reference Architecture](#)에서는 애플리케이션과 더 넓은 인터넷 간의 양방향 인터페이스를 보호하기 위해 인바운드, 아웃바운드 및 검사 VPC로 네트워크 계정을 설정할 것을 권장합니다.

중분 마이그레이션

한 번에 전체 전환을 수행하는 대신 애플리케이션을 조금씩 마이그레이션하는 전환 전략입니다. 예를 들어, 처음에는 소수의 마이크로서비스나 사용자만 새 시스템으로 이동할 수 있습니다. 모든 것

이 제대로 작동하는지 확인한 후에는 레거시 시스템을 폐기할 수 있을 때까지 추가 마이크로서비스 또는 사용자를 점진적으로 이동할 수 있습니다. 이 전략을 사용하면 대규모 마이그레이션과 관련된 위험을 줄일 수 있습니다.

Industry 4.0

[Klaus Schwab](#)이 연결성, 실시간 데이터, 자동화, 분석 및 AI/ML의 발전을 통한 제조 프로세스의 현대화를 지칭하기 위해 2016년 도입한 용어입니다.

인프라

애플리케이션의 환경 내에 포함된 모든 리소스와 자산입니다.

코드형 인프라(IaC)

구성 파일 세트를 통해 애플리케이션의 인프라를 프로비저닝하고 관리하는 프로세스입니다. IaC는 새로운 환경의 반복 가능성, 신뢰성 및 일관성을 위해 인프라 관리를 중앙 집중화하고, 리소스를 표준화하고, 빠르게 확장할 수 있도록 설계되었습니다.

산업용 사물 인터넷(IIoT)

제조, 에너지, 자동차, 의료, 생명과학, 농업 등의 산업 부문에서 인터넷에 연결된 센서 및 디바이스의 사용 자세한 내용은 [산업용 사물 인터넷\(IoT\) 디지털 트랜스포메이션 전략 구축](#)을 참조하십시오.

검사 VPC

AWS 다중 계정 아키텍처에서 VPC (동일하거나 AWS 리전다른), 인터넷 및 온프레미스 네트워크 간의 네트워크 트래픽 검사를 관리하는 중앙 집중식 VPC입니다. [AWS Security Reference Architecture](#)에서는 애플리케이션과 더 넓은 인터넷 간의 양방향 인터페이스를 보호하기 위해 인바운드, 아웃바운드 및 검사 VPC로 네트워크 계정을 설정할 것을 권장합니다.

사물 인터넷(IoT)

인터넷이나 로컬 통신 네트워크를 통해 다른 디바이스 및 시스템과 통신하는 센서 또는 프로세서가 내장된 연결된 물리적 객체의 네트워크 자세한 내용은 [IoT란?](#)을 참조하십시오.

해석력

모델의 예측이 입력에 따라 어떻게 달라지는지를 사람이 이해할 수 있는 정도를 설명하는 기계 학습 모델의 특성입니다. 자세한 내용은 [Machine learning model interpretability with AWS](#)를 참조하십시오.

IoT

[사물 인터넷을 참조하십시오.](#)

IT 정보 라이브러리(TIL)

IT 서비스를 제공하고 이러한 서비스를 비즈니스 요구 사항에 맞게 조정하기 위한 일련의 모범 사례 ITIL은 ITSM의 기반을 제공합니다.

IT 서비스 관리(TSM)

조직의 IT 서비스 설계, 구현, 관리 및 지원과 관련된 활동 클라우드 운영을 ITSM 도구와 통합하는 방법에 대한 자세한 내용은 [운영 통합 가이드](#)를 참조하십시오.

ITIL

[IT 정보 라이브러리를](#) 참조하십시오.

ITSM

[IT 서비스 관리를](#) 참조하십시오.

L

레이블 기반 액세스 제어(LBAC)

사용자 및 데이터 자체에 각각 보안 레이블 값을 명시적으로 할당하는 필수 액세스 제어(MAC)를 구현한 것입니다. 사용자 보안 레이블과 데이터 보안 레이블 간의 교차 부분에 따라 사용자가 볼 수 있는 행과 열이 결정됩니다.

랜딩 존

Landing Zone은 확장 가능하고 안전한 잘 설계된 다중 계정 AWS 환경입니다. 조직은 여기에서부터 보안 및 인프라 환경에 대한 확신을 가지고 워크로드와 애플리케이션을 신속하게 시작하고 배포할 수 있습니다. 랜딩 존에 대한 자세한 내용은 [안전하고 확장 가능한 다중 계정 AWS 환경 설정](#)을 참조하십시오.

대규모 마이그레이션

300대 이상의 서버 마이그레이션입니다.

LBAC

[레이블 기반 액세스 제어를](#) 참조하십시오.

최소 권한

작업을 수행하는 데 필요한 최소 권한을 부여하는 보안 모범 사례입니다. 자세한 내용은 IAM 설명서의 [최소 권한 적용](#)을 참조하십시오.

리프트 앤드 시프트

[7 R](#)을 참조하십시오.

리틀 엔디안 시스템

가장 덜 중요한 바이트를 먼저 저장하는 시스템입니다. [엔디안](#) 참조.

하위 환경

[환경 참조.](#)

M

기계 학습(ML)

패턴 인식 및 학습에 알고리즘과 기법을 사용하는 인공지능의 한 유형입니다. ML은 사물 인터넷 (IoT) 데이터와 같은 기록된 데이터를 분석하고 학습하여 패턴을 기반으로 통계 모델을 생성합니다. 자세한 내용은 [기계 학습](#)을 참조하십시오.

기본 브랜치

[브랜치](#) 참조.

악성 코드

컴퓨터 보안 또는 개인 정보를 침해하도록 설계된 소프트웨어 멀웨어는 컴퓨터 시스템을 방해하거나, 민감한 정보를 유출하거나, 무단 액세스를 얻을 수 있습니다. 멀웨어의 예로는 바이러스, 웜, 랜섬웨어, 트로이 목마, 스파이웨어, 키로거 등이 있습니다.

매니지드 서비스

AWS 서비스 인프라 계층, 운영 체제 및 플랫폼을 AWS 운영하며 사용자는 엔드포인트에 액세스하여 데이터를 저장하고 검색합니다. 관리형 서비스의 예로는 아마존 심플 스토리지 서비스 (Amazon S3) 와 아마존 DynamoDB가 있습니다. 이러한 서비스를 추상화된 서비스라고도 합니다.

제조 실행 시스템 (MES)

제조 현장에서 원자재를 완제품으로 전환하는 생산 프로세스를 추적, 모니터링, 문서화 및 제어하기 위한 소프트웨어 시스템입니다.

MAP

[Migration Acceleration 프로그램](#)을 참조하십시오.

기구

도구를 만들고 도구 채택을 유도한 다음 결과를 검토하여 조정하는 전체 프로세스입니다. 메커니즘은 작동하면서 자체적으로 강화되고 개선되는 사이클입니다. 자세한 내용은 [AWS Well-Architected 프레임워크에서의 메커니즘 구축을](#) 참조하십시오.

멤버 계정

조직의 일부인 관리 계정을 AWS 계정 제외한 모든 계정 AWS Organizations 하나의 계정은 한 번에 하나의 조직 멤버만 될 수 있습니다.

MES

[제조 실행 시스템을](#) 참조하십시오.

메시지 큐 텔레메트리 전송 (MQTT)

[퍼블리시/구독 패턴을 기반으로 하는 리소스가 제한된 IoT 디바이스를 위한 경량 machine-to-machine \(M2M\) 통신 프로토콜입니다.](#)

마이크로서비스

잘 정의된 API를 통해 통신하고 일반적으로 소규모 자체 팀이 소유하는 소규모 독립 서비스입니다. 예를 들어, 보험 시스템에는 영업, 마케팅 등의 비즈니스 역량이나 구매, 청구, 분석 등의 하위 영역에 매핑되는 마이크로 서비스가 포함될 수 있습니다. 마이크로서비스의 이점으로 민첩성, 유연한 확장, 손쉬운 배포, 재사용 가능한 코드, 복원력 등이 있습니다. [자세한 내용은 서버리스 서비스를 사용하여 마이크로서비스 통합을](#) 참조하십시오. [AWS](#)

마이크로서비스 아키텍처

각 애플리케이션 프로세스를 마이크로서비스로 실행하는 독립 구성 요소를 사용하여 애플리케이션을 구축하는 접근 방식입니다. 이러한 마이크로서비스는 경량 API를 사용하여 잘 정의된 인터페이스를 통해 통신합니다. 애플리케이션의 특정 기능에 대한 수요에 맞게 이 아키텍처의 각 마이크로 서비스를 업데이트, 배포 및 조정할 수 있습니다. 자세한 내용은 [마이크로서비스 구현을](#) 참조하십시오. [AWS](#)

Migration Acceleration Program(MAP)

조직이 클라우드로 전환하기 위한 강력한 운영 기반을 구축하고 초기 마이그레이션 비용을 상쇄할 수 있도록 컨설팅 지원, 교육 및 서비스를 제공하는 AWS 프로그램입니다. MAP에는 레거시 마이그레이션을 체계적인 방식으로 실행하기 위한 마이그레이션 방법론과 일반적인 마이그레이션 시나리오를 자동화하고 가속화하는 도구 세트가 포함되어 있습니다.

대규모 마이그레이션

애플리케이션 포트폴리오의 대다수를 웨이브를 통해 클라우드로 이동하는 프로세스로, 각 웨이브에서 더 많은 애플리케이션이 더 빠른 속도로 이동합니다. 이 단계에서는 이전 단계에서 배운 모범 사례와 교훈을 사용하여 팀, 도구 및 프로세스의 마이그레이션 팩토리를 구현하여 자동화 및 민첩한 제공을 통해 워크로드 마이그레이션을 간소화합니다. 이것은 [AWS 마이그레이션 전략](#)의 세 번째 단계입니다.

마이그레이션 팩토리

자동화되고 민첩한 접근 방식을 통해 워크로드 마이그레이션을 간소화하는 다기능 팀입니다. 마이그레이션 팩토리 팀에는 일반적으로 운영, 비즈니스 분석가 및 소유자, 마이그레이션 엔지니어, 개발자 및 스프린트에서 일하는 DevOps 전문가가 포함됩니다. 엔터프라이즈 애플리케이션 포트폴리오의 20~50%는 공장 접근 방식으로 최적화할 수 있는 반복되는 패턴으로 구성되어 있습니다. 자세한 내용은 이 콘텐츠 세트의 [클라우드 마이그레이션 팩토리 가이드](#)와 [마이그레이션 팩토리에 대한 설명](#)을 참조하십시오.

마이그레이션 메타데이터

마이그레이션을 완료하는 데 필요한 애플리케이션 및 서버에 대한 정보 각 마이그레이션 패턴에는 서로 다른 마이그레이션 메타데이터 세트가 필요합니다. 마이그레이션 메타데이터의 예로는 대상 서브넷, 보안 그룹, 계정 등이 있습니다. AWS

마이그레이션 패턴

사용되는 마이그레이션 전략, 마이그레이션 대상, 마이그레이션 애플리케이션 또는 서비스를 자세히 설명하는 반복 가능한 마이그레이션 작업입니다. 예: 애플리케이션 마이그레이션 서비스를 사용하여 Amazon EC2로 AWS 마이그레이션을 재호스팅합니다.

Migration Portfolio Assessment(MPA)

로 마이그레이션하기 위한 비즈니스 사례를 검증하기 위한 정보를 제공하는 온라인 도구입니다. AWS 클라우드 MPA는 상세한 포트폴리오 평가(서버 적정 규모 조정, 가격 책정, TCO 비교, 마이그레이션 비용 분석)와 마이그레이션 계획(애플리케이션 데이터 분석 및 데이터 수집, 애플리케이션 그룹화, 마이그레이션 우선순위 지정, 웨이브 계획)을 제공합니다. [MPA 도구](#) (로그인 필요) 는 모든 컨설턴트와 APN 파트너 AWS 컨설턴트에게 무료로 제공됩니다.

마이그레이션 준비 상태 평가(MRA)

CAF를 사용하여 조직의 클라우드 준비 상태에 대한 통찰력을 얻고, 강점과 약점을 파악하고, 식별된 격차를 해소하기 위한 실행 계획을 수립하는 프로세스입니다. AWS 자세한 내용은 [마이그레이션 준비 가이드](#)를 참조하십시오. MRA는 [AWS 마이그레이션 전략](#)의 첫 번째 단계입니다.

마이그레이션 전략

워크로드를 로 마이그레이션하는 데 사용된 접근 방식. AWS 클라우드자세한 내용은 이 용어집의 [7R 항목](#) 및 [대규모 마이그레이션 가속화를 위한 조직 동원을 참조하십시오.](#)

ML

[기계 학습을 참조하십시오.](#)

현대화

비용을 절감하고 효율성을 높이고 혁신을 활용하기 위해 구식(레거시 또는 모놀리식) 애플리케이션과 해당 인프라를 클라우드의 민첩하고 탄력적이고 가용성이 높은 시스템으로 전환하는 것입니다. 자세한 내용은 [의 AWS 클라우드애플리케이션 현대화 전략](#)을 참조하십시오.

현대화 준비 상태 평가

조직 애플리케이션의 현대화 준비 상태를 파악하고, 이점, 위험 및 종속성을 식별하고, 조직이 해당 애플리케이션의 향후 상태를 얼마나 잘 지원할 수 있는지를 확인하는 데 도움이 되는 평가입니다. 평가 결과는 대상 아키텍처의 청사진, 현대화 프로세스의 개발 단계와 마일스톤을 자세히 설명하는 로드맵 및 파악된 격차를 해소하기 위한 실행 계획입니다. 자세한 내용은 [에서 애플리케이션의 현대화 준비 상태 평가를 참조하십시오.](#) AWS 클라우드

모놀리식 애플리케이션(모놀리식 유형)

긴밀하게 연결된 프로세스를 사용하여 단일 서비스로 실행되는 애플리케이션입니다. 모놀리식 애플리케이션에는 몇 가지 단점이 있습니다. 한 애플리케이션 기능에 대한 수요가 급증하면 전체 아키텍처 규모를 조정해야 합니다. 코드 베이스가 커지면 모놀리식 애플리케이션의 기능을 추가하거나 개선하는 것도 더 복잡해집니다. 이러한 문제를 해결하기 위해 마이크로서비스 아키텍처를 사용할 수 있습니다. 자세한 내용은 [마이크로서비스로 모놀리식 유형 분해](#)를 참조하십시오.

MPA

[마이그레이션 포트폴리오 평가](#)를 참조하십시오.

MQTT

[메시지 큐 원격 분석](#) 전송을 참조하십시오.

멀티클래스 분류

여러 클래스에 대한 예측(2개 이상의 결과 중 하나 예측)을 생성하는 데 도움이 되는 프로세스입니다. 예를 들어, ML 모델이 '이 제품은 책인가요, 자동차인가요, 휴대폰인가요?' 또는 '이 고객이 가장 관심을 갖는 제품 범주는 무엇인가요?'라고 물을 수 있습니다.

변경 가능한 인프라

프로덕션 워크로드를 위해 기존 인프라를 업데이트하고 수정하는 모델입니다. 일관성, 안정성 및 예측 가능성을 개선하기 위해 AWS Well-Architected Framework는 [변경 불가능한](#) 인프라를 모범 사례로 사용할 것을 권장합니다.

O

OAC

[원본 액세스 제어를 참조하십시오.](#)

좋아요

[원본 액세스 ID를 참조하십시오.](#)

OCM

[조직 변경 관리를 참조하십시오.](#)

오프라인 마이그레이션

마이그레이션 프로세스 중 소스 워크로드가 중단되는 마이그레이션 방법입니다. 이 방법은 가동 중지 증가를 수반하며 일반적으로 작고 중요하지 않은 워크로드에 사용됩니다.

O

[운영 통합을 참조하십시오.](#)

안녕하세요.

[운영 수준 계약을 참조하십시오.](#)

온라인 마이그레이션

소스 워크로드를 오프라인 상태로 전환하지 않고 대상 시스템에 복사하는 마이그레이션 방법입니다. 워크로드에 연결된 애플리케이션은 마이그레이션 중에도 계속 작동할 수 있습니다. 이 방법은 가동 중지 차단 또는 최소화를 수반하며 일반적으로 중요한 프로덕션 워크로드에 사용됩니다.

OPC-UA

[오픈 프로세스 커뮤니케이션 - 통합](#) 아키텍처를 참조하십시오.

오픈 프로세스 커뮤니케이션 - 통합 아키텍처 (OPC-UA)

산업 machine-to-machine 자동화를 위한 (M2M) 통신 프로토콜. OPC-UA는 데이터 암호화, 인증 및 권한 부여 체계와 함께 상호 운용성 표준을 제공합니다.

운영 수준 협약(OLA)

서비스 수준에 관한 계약(SLA)을 지원하기 위해 직무 IT 그룹이 서로에게 제공하기로 약속한 내용을 명확히 하는 계약입니다.

운영 준비 검토 (ORR)

인시던트 및 발생 가능한 실패의 범위를 이해, 평가, 예방 또는 줄이는 데 도움이 되는 질문 및 관련 모범 사례로 구성된 체크리스트입니다. 자세한 내용은 Well-Architected AWS 프레임워크의 [운영 준비 상태 검토 \(ORR\)](#) 를 참조하십시오.

운영 기술 (OT)

물리적 환경과 함께 작동하여 산업 운영, 장비 및 인프라를 제어하는 하드웨어 및 소프트웨어 시스템. 제조 분야에서는 OT와 정보 기술 (IT) 시스템의 통합이 [인더스트리 4.0](#) 혁신의 핵심 초점입니다.

운영 통합(OI)

클라우드에서 운영을 현대화하는 프로세스로 준비 계획, 자동화 및 통합을 수반합니다. 자세한 내용은 [운영 통합 가이드](#)를 참조하십시오.

조직 트레일

이를 통해 AWS CloudTrail 생성되는 트레일은 조직 AWS 계정 내 모든 사용자의 모든 이벤트를 기록합니다. AWS Organizations이 트레일은 조직에 속한 각 AWS 계정에 생성되고 각 계정의 활동을 추적합니다. 자세한 내용은 CloudTrail 설명서에서 [조직을 위한 트레일 만들기를](#) 참조하십시오.

조직 변경 관리(OCM)

사람, 문화 및 리더십 관점에서 중대하고 파괴적인 비즈니스 혁신을 관리하기 위한 프레임워크입니다. OCM은 변화 채택을 가속화하고, 과도기적 문제를 해결하고, 문화 및 조직적 변화를 주도함으로써 조직이 새로운 시스템 및 전략을 준비하고 전환할 수 있도록 지원합니다. 클라우드 채택 프로젝트에 필요한 변화 속도 때문에 AWS 마이그레이션 전략에서는 이 프레임워크를 사용자 가속화라고 합니다. 자세한 내용은 [사용 가이드](#)를 참조하십시오.

오리진 액세스 제어(OAC)

CloudFront에서는 Amazon Simple Storage Service (Amazon S3) 콘텐츠의 보안을 위해 액세스를 제한하는 향상된 옵션을 제공합니다. OAC는 모든 S3 버킷 AWS 리전, AWS KMS (SSE-KMS) 를 사용한 서버 측 암호화, S3 버킷에 대한 동적 및 요청을 모두 지원합니다. PUT DELETE

오리진 액세스 ID(OAI)

CloudFront에서는 Amazon S3 콘텐츠 보안을 위해 액세스를 제한하는 옵션입니다. OAI를 사용하면 Amazon S3가 인증할 수 있는 보안 주체를 CloudFront 생성합니다. 인증된 보안 주체는 특정 배

포를 통해서만 S3 버킷의 콘텐츠에 액세스할 수 있습니다. CloudFront 더 세분화되고 향상된 액세스 제어를 제공하는 [OAC](#)도 참조하십시오.

또는

[운영 준비 상태](#) 검토를 참조하십시오.

아니요

[운영 기술을](#) 참조하십시오.

아웃바운드(송신) VPC

AWS 다중 계정 아키텍처에서 애플리케이션 내에서 시작되는 네트워크 연결을 처리하는 VPC입니다. [AWS Security Reference Architecture](#)에서는 애플리케이션과 더 넓은 인터넷 간의 양방향 인터페이스를 보호하기 위해 인바운드, 아웃바운드 및 검사 VPC로 네트워크 계정을 설정할 것을 권장합니다.

P

권한 경계

사용자나 역할이 가질 수 있는 최대 권한을 설정하기 위해 IAM 보안 주체에 연결되는 IAM 관리 정책입니다. 자세한 내용은 IAM 설명서의 [권한 경계](#)를 참조하십시오.

개인 식별 정보(PII)

직접 보거나 다른 관련 데이터와 함께 짝을 지을 때 개인의 신원을 합리적으로 추론하는 데 사용할 수 있는 정보입니다. PII의 예로는 이름, 주소, 연락처 정보 등이 있습니다.

PII

[개인 식별](#) 정보를 참조하십시오.

플레이북

클라우드에서 핵심 운영 기능을 제공하는 등 마이그레이션과 관련된 작업을 캡처하는 일련의 사전 정의된 단계입니다. 플레이북은 스크립트, 자동화된 런북 또는 현대화된 환경을 운영하는 데 필요한 프로세스나 단계 요약의 형태를 취할 수 있습니다.

PLC

[프로그래머블 로직 컨트롤러](#)를 참조하십시오.

PLM

[제품 라이프사이클 관리](#)를 참조하십시오.

정책

권한을 정의 ([ID 기반 정책 참조](#)) 하거나, 액세스 조건을 지정 ([리소스 기반 정책 참조](#)) 하거나, 조직 내 모든 계정에 대한 최대 권한을 정의 AWS Organizations ([서비스 제어 정책 참조](#)) 할 수 있는 개체입니다.

다국어 지속성

데이터 액세스 패턴 및 기타 요구 사항을 기반으로 독립적으로 마이크로서비스의 데이터 스토리지 기술 선택. 마이크로서비스가 동일한 데이터 스토리지 기술을 사용하는 경우 구현 문제가 발생하거나 성능이 저하될 수 있습니다. 요구 사항에 가장 적합한 데이터 스토어를 사용하면 마이크로서비스를 더 쉽게 구현하고 성능과 확장성을 높일 수 있습니다. 자세한 내용은 [마이크로서비스에서 데이터 지속성 활성화](#)를 참조하십시오.

포트폴리오 평가

마이그레이션을 계획하기 위해 애플리케이션 포트폴리오를 검색 및 분석하고 우선순위를 정하는 프로세스입니다. 자세한 내용은 [마이그레이션 준비 상태 평가](#)를 참조하십시오.

조건자

일반적으로 조항에 있는 true false OR를 반환하는 쿼리 조건입니다. WHERE

조건부 푸시다운

전송하기 전에 쿼리의 데이터를 필터링하는 데이터베이스 쿼리 최적화 기법입니다. 이렇게 하면 관계형 데이터베이스에서 검색하고 처리해야 하는 데이터의 양이 줄어들고 쿼리 성능이 향상됩니다.

예방적 제어

이벤트 발생을 방지하도록 설계된 보안 제어입니다. 이 제어는 네트워크에 대한 무단 액세스나 원치 않는 변경을 방지하는 데 도움이 되는 1차 방어선입니다. 자세한 내용은 Implementing security controls on AWS의 [Preventative controls](#)를 참조하십시오.

보안 주체

작업을 수행하고 리소스에 액세스할 수 있는 AWS 있는 엔티티 이 엔티티는 일반적으로 IAM 역할의 루트 사용자 또는 사용자입니다. AWS 계정자세한 내용은 IAM 설명서의 [역할 용어 및 개념](#)의 보안 주체를 참조하십시오.

개인 정보 보호 중심 설계

전체 엔지니어링 프로세스에서 개인 정보를 고려하는 시스템 엔지니어링에서의 접근 방식입니다.

프라이빗 호스팅 영역

Amazon Route 53에서 하나 이상의 VPC 내 도메인과 하위 도메인에 대한 DNS 쿼리에 응답하는 방법에 대한 정보가 담긴 컨테이너입니다. 자세한 내용은 Route 53 설명서의 [프라이빗 호스팅 영역 작업](#)을 참조하십시오.

사전 예방 제어

규정을 준수하지 않는 리소스의 배포를 방지하도록 설계된 [보안 제어입니다](#). 이러한 컨트롤은 리소스를 프로비저닝하기 전에 리소스를 스캔합니다. 리소스가 컨트롤과 호환되지 않으면 프로비저닝되지 않습니다. 자세한 내용은 AWS Control Tower 설명서의 [컨트롤 참조 안내서](#)를 참조하고 보안 제어 구현의 [사전 제어를](#) 참조하십시오. AWS

제품 라이프사이클 관리 (PLM)

설계, 개발, 출시부터 성장 및 성숙도, 폐기 및 제거에 이르는 전체 라이프사이클에 걸쳐 제품에 대한 데이터 및 프로세스를 관리하는 것입니다.

프로덕션 환경

[환경](#)을 참조하십시오.

프로그래머블 로직 컨트롤러 (PLC)

제조 분야에서 기계를 모니터링하고 제조 프로세스를 자동화하는 매우 안정적이고 적응력이 뛰어난 컴퓨터입니다.

가명화

데이터세트의 개인 식별자를 자리 표시자 값으로 바꾸는 프로세스입니다. 가명화는 개인 정보를 보호하는 데 도움이 될 수 있습니다. 가명화된 데이터는 여전히 개인 데이터로 간주됩니다.

게시/구독 (게시/구독)

마이크로서비스 간의 비동기 통신을 통해 확장성과 응답성을 개선할 수 있는 패턴입니다. 예를 들어 마이크로서비스 기반 [MES에서](#) 마이크로서비스는 다른 마이크로서비스가 구독할 수 있는 채널에 이벤트 메시지를 게시할 수 있습니다. 시스템은 게시 서비스를 변경하지 않고도 새 마이크로서비스를 추가할 수 있습니다.

Q

쿼리 계획

SQL 관계형 데이터베이스 시스템의 데이터에 액세스하는 데 사용되는 일련의 단계 (예: 지침).

쿼리 계획 회귀

데이터베이스 서비스 최적화 프로그램이 데이터베이스 환경을 변경하기 전보다 덜 최적의 계획을 선택하는 경우입니다. 통계, 제한 사항, 환경 설정, 쿼리 파라미터 바인딩 및 데이터베이스 엔진 업데이트의 변경으로 인해 발생할 수 있습니다.

R

RACI 매트릭스

RACI ([책임, 책임, 상담, 정보 제공](#)) 를 참조하십시오.

랜섬웨어

결제 완료될 때까지 컴퓨터 시스템이나 데이터에 대한 액세스를 차단하도록 설계된 악성 소프트웨어입니다.

RASCI 매트릭스

[책임, 책임, 상담, 정보 제공 \(RACI\)](#) 을 참조하십시오.

RCAC

[행 및 열 액세스 제어를](#) 참조하십시오.

읽기 전용 복제본

읽기 전용 용도로 사용되는 데이터베이스의 사본입니다. 쿼리를 읽기 전용 복제본으로 라우팅하여 기본 데이터베이스의 로드를 줄일 수 있습니다.

재설계

[7 R](#)을 참조하십시오.

Recovery Point Objective(RPO)

마지막 데이터 복구 시점 이후 허용되는 최대 시간입니다. 이에 따라 마지막 복구 시점과 서비스 중단 사이에 허용되는 데이터 손실로 간주되는 범위가 결정됩니다.

Recovery Time Objective(RTO)

서비스 중단과 서비스 복원 사이의 허용 가능한 지연 시간입니다.

리팩터링

[7 R](#)을 참조하십시오.

리전

지리적 AWS 영역별 리소스 모음. AWS 리전 각각은 격리되어 있고 서로 독립적이므로 내결함성, 안정성 및 복원력을 제공합니다. 자세한 내용은 [사용할 수 있는 AWS 리전 계정 지정을](#) 참조하십시오.

회귀

숫자 값을 예측하는 ML 기법입니다. 예를 들어, '이 집은 얼마에 팔릴까?'라는 문제를 풀기 위해 ML 모델은 선형 회귀 모델을 사용하여 주택에 대해 알려진 사실(예: 면적)을 기반으로 주택의 매매 가격을 예측할 수 있습니다.

리호스팅

[7 R을](#) 참조하십시오.

release

배포 프로세스에서 변경 사항을 프로덕션 환경으로 승격시키는 행위입니다.

고쳐 놓다

[7 R을](#) 참조하십시오.

리플랫폼

[7 R을](#) 참조하십시오.

환매

[7 R을](#) 참조하십시오.

복원력

장애를 견디거나 장애를 복구할 수 있는 애플리케이션의 능력 [고가용성](#) 및 [재해 복구](#)는 복원력을 계획할 때 일반적으로 고려해야 할 사항입니다. AWS 클라우드 자세한 내용은 [AWS 클라우드 복원력을](#) 참조하십시오.

리소스 기반 정책

Amazon S3 버킷, 엔드포인트, 암호화 키 등의 리소스에 연결된 정책입니다. 이 유형의 정책은 액세스가 허용된 보안 주체, 지원되는 작업 및 충족해야 하는 기타 조건을 지정합니다.

RACI(Responsible, Accountable, Consulted, Informed) 매트릭스

마이그레이션 활동 및 클라우드 운영에 참여하는 모든 당사자의 역할과 책임을 정의하는 매트릭스입니다. 매트릭스 이름은 매트릭스에 정의된 책임 유형에서 파생됩니다. 실무 담당자 (R), 의사 결

정권자 (A), 업무 수행 조연자 (C), 결과 통보 대상자 (I). 지원자는 (S) 선택사항입니다. 지원자를 포함하면 매트릭스를 RASCI 매트릭스라고 하고, 지원자를 제외하면 RACI 매트릭스라고 합니다.

대응 제어

보안 기준에서 벗어나거나 부정적인 이벤트를 해결하도록 설계된 보안 제어입니다. 자세한 내용은 Implementing security controls on AWS의 [Responsive controls](#)를 참조하십시오.

retain

[7 R](#)을 참조하십시오.

은퇴

[7 R](#)을 참조하십시오.

회전

공격자가 자격 증명에 액세스하는 것을 더 어렵게 만들기 위해 [암호](#)를 주기적으로 업데이트하는 프로세스입니다.

행 및 열 액세스 제어(RCAC)

액세스 규칙이 정의된 기본적이고 유연한 SQL 표현식을 사용합니다. RCAC는 행 권한과 열 마스크로 구성됩니다.

RPO

[복구 지점 목표를](#) 참조하십시오.

RTO

[복구 시간 목표를](#) 참조하십시오.

런복

특정 작업을 수행하는 데 필요한 일련의 수동 또는 자동 절차입니다. 일반적으로 오류율이 높은 반복 작업이나 절차를 간소화하기 위해 런복을 만듭니다.

S

SAML 2.0

많은 ID 제공업체 (IdPs) 가 사용하는 개방형 표준입니다. 이 기능을 사용하면 페더레이션 싱글 사인온 (SSO) 이 가능하므로 조직의 모든 사용자를 위해 IAM에서 사용자를 생성하지 않고도 사용자가 AWS API 작업에 AWS Management Console 로그인하거나 API 작업을 호출할 수 있습니다.

SAML 2.0 기반 페더레이션에 대한 자세한 내용은 IAM 설명서의 [SAML 2.0 기반 페더레이션 정보](#)를 참조하십시오.

SCADA

[감독 제어 및 데이터 수집](#)을 참조하십시오.

SCP

[서비스 제어 정책](#)을 참조하십시오.

secret

에는 AWS Secrets Manager 암호화된 형태로 저장하는 비밀번호나 사용자 자격 증명과 같은 기밀 또는 제한된 정보. 비밀 값과 해당 메타데이터로 구성됩니다. 비밀 값은 바이너리, 단일 문자열 또는 여러 문자열일 수 있습니다. 자세한 내용은 [Secrets Manager 시크릿에는 무엇이 들어 있나요?](#)를 참조하십시오. Secrets Manager 설명서에서 확인할 수 있습니다.

보안 제어

위험 행위자가 보안 취약성을 악용하는 능력을 방지, 탐지 또는 감소시키는 기술적 또는 관리적 가드레일입니다. [보안 제어에는 예방적, 탐정적, 대응적, 사전 예방적 제어의 네 가지 기본 유형이 있습니다.](#)

보안 강화

공격 표면을 줄여 공격에 대한 저항력을 높이는 프로세스입니다. 더 이상 필요하지 않은 리소스 제거, 최소 권한 부여의 보안 모범 사례 구현, 구성 파일의 불필요한 기능 비활성화 등의 작업이 여기에 포함될 수 있습니다.

보안 정보 및 이벤트 관리(SIEM) 시스템

보안 정보 관리(SIM)와 보안 이벤트 관리(SEM) 시스템을 결합하는 도구 및 서비스입니다. SIEM 시스템은 서버, 네트워크, 디바이스 및 기타 소스에서 데이터를 수집, 모니터링 및 분석하여 위협과 보안 침해를 탐지하고 알림을 생성합니다.

보안 대응 자동화

보안 이벤트에 자동으로 대응하거나 보안 이벤트를 해결하도록 설계된 사전 정의되고 프로그래밍된 조치입니다. 이러한 자동화는 보안 모범 사례를 구현하는 데 도움이 되는 [탐지](#) 또는 [대응형](#) 보안 제어 역할을 합니다. AWS 자동 응답 조치의 예로는 VPC 보안 그룹 수정, Amazon EC2 인스턴스 패치, 자격 증명 교체 등이 있습니다.

서버 측 암호화

수신자에 의한 목적지의 데이터 암호화 AWS 서비스

서비스 제어 정책(SCP)

AWS Organizations에 속한 조직의 모든 계정에 대한 권한을 중앙 집중식으로 제어하는 정책입니다. SCP는 관리자가 사용자 또는 역할에 위임할 수 있는 작업에 대해 제한을 설정하거나 가드레일을 정의합니다. SCP를 허용 목록 또는 거부 목록으로 사용하여 허용하거나 금지할 서비스 또는 작업을 지정할 수 있습니다. 자세한 내용은 AWS Organizations 설명서의 [서비스 제어 정책을](#) 참조하십시오.

서비스 엔드포인트

의 진입점 URL입니다 AWS 서비스. 엔드포인트를 사용하여 대상 서비스에 프로그래밍 방식으로 연결할 수 있습니다. 자세한 내용은 AWS 일반 참조의 [AWS 서비스 엔드포인트](#)를 참조하십시오.

서비스 수준에 관한 계약(SLA)

IT 팀이 고객에게 제공하기로 약속한 내용(예: 서비스 가동 시간 및 성능)을 명시한 계약입니다.

서비스 수준 표시기 (SLI)

오류율, 가용성 또는 처리량과 같은 서비스의 성능 측면을 측정하는 것입니다.

서비스 수준 목표 (SLO)

[서비스 수준 지표로 측정되는 서비스 상태를 나타내는 대상 지표입니다.](#)

공동 책임 모델

클라우드 보안 및 규정 준수에 AWS 대한 책임을 공유하는 것을 설명하는 모델입니다. AWS 클라우드의 보안을 책임지는 반면, 사용자는 클라우드에서의 보안을 담당합니다. 자세한 내용은 [공동 책임 모델](#)을 참조하십시오.

시앰

[보안 정보 및 이벤트 관리 시스템을](#) 참조하십시오.

단일 장애 지점 (SPOF)

응용 프로그램의 중요한 단일 구성 요소에서 발생한 오류로 인해 시스템이 중단될 수 있습니다.

SLA

SLA ([서비스 수준 계약](#)) 를 참조하십시오.

SLI

[서비스 수준 표시기](#) 참조.

SLO

[서비스 수준 목표를](#) 참조하십시오.

split-and-seed 모델

현대화 프로젝트를 확장하고 가속화하기 위한 패턴입니다. 새로운 기능과 제품 릴리스가 정의되면 핵심 팀이 분할되어 새로운 제품 팀이 만들어집니다. 이를 통해 조직의 역량과 서비스 규모를 조정하고, 개발자 생산성을 개선하고, 신속한 혁신을 지원할 수 있습니다. 자세한 내용은 [의 애플리케이션 현대화를 위한 단계별 접근 방식을 참조하십시오. AWS 클라우드](#)

SPOF

[단일 장애 지점 보기.](#)

스타 스키마

하나의 큰 팩트 테이블을 사용하여 트랜잭션 또는 측정 데이터를 저장하고 하나 이상의 작은 차원 테이블을 사용하여 데이터 속성을 저장하는 데이터베이스 구성 구조입니다. 이 구조는 [데이터 웨어하우스에서](#) 사용하거나 비즈니스 인텔리전스 용도로 설계되었습니다.

Strangler Fig 패턴

레거시 시스템을 폐기할 수 있을 때까지 시스템 기능을 점진적으로 다시 작성하고 교체하여 모놀리식 시스템을 현대화하기 위한 접근 방식. 이 패턴은 무화과 덩굴이 나무로 자라 결국 숙주를 압도하고 대체하는 것과 비슷합니다. [Martin Fowler](#)가 모놀리식 시스템을 다시 작성할 때 위험을 관리하는 방법으로 이 패턴을 도입했습니다. 이 패턴을 적용하는 방법의 예는 [컨테이너 및 Amazon API Gateway를 사용하여 기존의 Microsoft ASP.NET\(ASMX\) 웹 서비스를 점진적으로 현대화하는 방법을 참조하십시오.](#)

서브넷

VPC의 IP 주소 범위입니다. 서브넷은 단일 가용 영역에 상주해야 합니다.

감독 통제 및 데이터 수집 (SCADA)

제조 시 하드웨어와 소프트웨어를 사용하여 물리적 자산과 생산 작업을 모니터링하는 시스템입니다.

대칭 암호화

동일한 키를 사용하여 데이터를 암호화하고 복호화하는 암호화 알고리즘입니다.

합성 테스트

잠재적 문제를 감지하거나 성능을 모니터링하기 위해 사용자 상호 작용을 시뮬레이션하는 방식으로 시스템을 테스트합니다. [Amazon CloudWatch Synthetics](#)를 사용하여 이러한 테스트를 생성할 수 있습니다.

T

tags

리소스 구성을 위한 메타데이터 역할을 하는 키-값 쌍. AWS 태그를 사용하면 리소스를 손쉽게 관리, 식별, 정리, 검색 및 필터링할 수 있습니다. 자세한 내용은 [AWS 리소스에 태그 지정](#)을 참조하십시오.

대상 변수

지도 ML에서 예측하려는 값으로, 결과 변수라고도 합니다. 예를 들어, 제조 설정에서 대상 변수는 제품 결함일 수 있습니다.

작업 목록

런북을 통해 진행 상황을 추적하는 데 사용되는 도구입니다. 작업 목록에는 런북의 개요와 완료해야 할 일반 작업 목록이 포함되어 있습니다. 각 일반 작업에 대한 예상 소요 시간, 소유자 및 진행 상황이 작업 목록에 포함됩니다.

테스트 환경

[환경을 참조하십시오.](#)

훈련

ML 모델이 학습할 수 있는 데이터를 제공하는 것입니다. 훈련 데이터에는 정답이 포함되어야 합니다. 학습 알고리즘은 훈련 데이터에서 대상(예측하려는 답)에 입력 데이터 속성을 매핑하는 패턴을 찾고, 이러한 패턴을 캡처하는 ML 모델을 출력합니다. 그런 다음 ML 모델을 사용하여 대상을 모르는 새 데이터에 대한 예측을 할 수 있습니다.

전송 게이트웨이

VPC와 온프레미스 네트워크를 상호 연결하는 데 사용할 수 있는 네트워크 전송 허브입니다. 자세한 내용은 AWS Transit Gateway 설명서의 [트랜짓 게이트웨이란 무엇입니까?](#)를 참조하십시오.

트렁크 기반 워크플로

개발자가 기능 브랜치에서 로컬로 기능을 구축하고 테스트한 다음 해당 변경 사항을 기본 브랜치에 병합하는 접근 방식입니다. 이후 기본 브랜치는 개발, 프로덕션 이전 및 프로덕션 환경에 순차적으로 구축됩니다.

신뢰할 수 있는 액세스

조직 내 AWS Organizations 및 해당 계정에서 사용자를 대신하여 작업을 수행하도록 지정한 서비스에 권한 부여 신뢰할 수 있는 서비스는 필요할 때 각 계정에 서비스 연결 역할을 생성하여 관

리 작업을 수행합니다. 자세한 내용은 AWS Organizations 설명서의 [다른 AWS 서비스와 AWS Organizations 함께 사용](#)을 참조하십시오.

튜닝

ML 모델의 정확도를 높이기 위해 훈련 프로세스의 측면을 여러 변경하는 것입니다. 예를 들어, 레이블링 세트를 생성하고 레이블을 추가한 다음 다양한 설정에서 이러한 단계를 여러 번 반복하여 모델을 최적화하는 방식으로 ML 모델을 훈련할 수 있습니다.

피자 두 판 팀

피자 두 판만 들고 배블리 먹을 수 있는 소규모 DevOps 팀. 피자 두 판 팀 규모는 소프트웨어 개발에 있어 가능한 최상의 공동 작업 기회를 보장합니다.

U

불확실성

예측 ML 모델의 신뢰성을 저해할 수 있는 부정확하거나 불완전하거나 알려지지 않은 정보를 나타내는 개념입니다. 불확실성에는 두 가지 유형이 있습니다. 인식론적 불확실성은 제한적이고 불완전한 데이터에 의해 발생하는 반면, 우연한 불확실성은 데이터에 내재된 노이즈와 무작위성에 의해 발생합니다. 자세한 내용은 [Quantifying uncertainty in deep learning systems](#) 가이드를 참조하십시오.

차별화되지 않은 작업

애플리케이션을 만들고 운영하는 데 필요하지만 최종 사용자에게 직접적인 가치를 제공하거나 경쟁 우위를 제공하지 못하는 작업을 헤비 리프팅이라고도 합니다. 차별화되지 않은 작업의 예로는 조달, 유지보수, 용량 계획 등이 있습니다.

상위 환경

[환경을](#) 보세요.

V

정리

스토리지를 회수하고 성능을 향상시키기 위해 증분 업데이트 후 정리 작업을 수반하는 데이터베이스 유지 관리 작업입니다.

버전 제어

리포지토리의 소스 코드 변경과 같은 변경 사항을 추적하는 프로세스 및 도구입니다.

VPC 피어링

프라이빗 IP 주소를 사용하여 트래픽을 라우팅할 수 있게 하는 두 VPC 간의 연결입니다. 자세한 내용은 Amazon VPC 설명서의 [VPC 피어링이란?](#)을 참조하십시오.

취약성

시스템 보안을 손상시키는 소프트웨어 또는 하드웨어 결함입니다.

W

웹 캐시

자주 액세스하는 최신 관련 데이터를 포함하는 버퍼 캐시입니다. 버퍼 캐시에서 데이터베이스 인스턴스를 읽을 수 있기 때문에 주 메모리나 디스크에서 읽는 것보다 빠릅니다.

웜 데이터

자주 액세스하지 않는 데이터입니다. 이런 종류의 데이터를 쿼리할 때는 일반적으로 적절히 느린 쿼리가 허용됩니다.

윈도우 함수

현재 레코드와 어떤 식으로든 관련된 행 그룹에 대해 계산을 수행하는 SQL 함수입니다. 윈도우 함수는 이동 평균을 계산하거나 현재 행의 상대적 위치를 기반으로 행 값에 액세스하는 등의 작업을 처리하는 데 유용합니다.

워크로드

고객 대면 애플리케이션이나 백엔드 프로세스 같이 비즈니스 가치를 창출하는 리소스 및 코드 모음입니다.

워크스트림

마이그레이션 프로젝트에서 특정 작업 세트를 담당하는 직무 그룹입니다. 각 워크스트림은 독립적이지만 프로젝트의 다른 워크스트림을 지원합니다. 예를 들어, 포트폴리오 워크스트림은 애플리케이션 우선순위 지정, 웨이브 계획, 마이그레이션 메타데이터 수집을 담당합니다. 포트폴리오 워크스트림은 이러한 자산을 마이그레이션 워크스트림에 전달하고, 마이그레이션 워크스트림은 서버와 애플리케이션을 마이그레이션합니다.

원

한 번 쓰고, 많이 읽으세요.

WQF

AWS 워크로드 검증 프레임워크를 참조하십시오.

한 번 작성하고 여러 번 읽기 (WORM)

데이터를 한 번 쓰고 데이터가 삭제되거나 수정되지 않도록 하는 스토리지 모델입니다. 인증된 사용자는 필요한 만큼 데이터를 여러 번 읽을 수 있지만 변경할 수는 없습니다. 이 데이터 스토리지 인프라는 변경할 수 없는 것으로 간주됩니다.

Z

제로데이 익스플로잇

제로데이 취약점을 악용하는 공격 (일반적으로 멀웨어)입니다.

제로데이 취약성

프로덕션 시스템의 명백한 결함 또는 취약성입니다. 위협 행위자는 이러한 유형의 취약성을 사용하여 시스템을 공격할 수 있습니다. 개발자는 공격의 결과로 취약성을 인지하는 경우가 많습니다.

좀비 애플리케이션

평균 CPU 및 메모리 사용량이 5% 미만인 애플리케이션입니다. 마이그레이션 프로젝트에서는 이러한 애플리케이션을 사용 중지하는 것이 일반적입니다.

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.