



Padrões, arquiteturas e implementações de design de nuvem

AWS Orientação prescritiva



AWS Orientação prescritiva: Padrões, arquiteturas e implementações de design de nuvem

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

As marcas comerciais e imagens de marcas da Amazon não podem ser usadas no contexto de nenhum produto ou serviço que não seja da Amazon, nem de qualquer maneira que possa gerar confusão entre os clientes ou que deprecie ou desprestige a Amazon. Todas as outras marcas comerciais que não pertencem à Amazon pertencem a seus respectivos proprietários, que podem ou não ser afiliados, patrocinados pela Amazon ou ter conexão com ela.

Table of Contents

Introdução	1
Resultados de negócios desejados	2
Padrão da camada anticorrupção	3
Intenção	3
Motivação	3
Aplicabilidade	3
Problemas e considerações	4
Implementação	5
Arquitetura de alto nível	5
Implementação usando serviços AWS	6
Código de exemplo	7
Repositório GitHub	8
Conteúdo relacionado	9
Padrões de roteamento de API	10
Roteamento de nome de host	10
Caso de uso típico	10
Prós	11
Contras	11
Roteamento de caminhos	12
Caso de uso típico	12
Proxy reverso de serviço HTTP	12
API Gateway	14
CloudFront	16
Roteamento de cabeçalho HTTP	17
Prós	18
Contras	18
Padrão de disjuntor	19
Intenção	19
Motivação	19
Aplicabilidade	20
Problemas e considerações	20
Implementação	21
Arquitetura de alto nível	21
Implementação usando AWS serviços	22

Código de exemplo	23
GitHub repositório	24
Referências do blog	25
Conteúdo relacionado	25
Padrão de fornecimento de eventos	26
Intenção	26
Motivação	26
Aplicabilidade	26
Problemas e considerações	27
Implementação	29
Arquitetura de alto nível	29
Implementação usando serviços da AWS	31
Referências do blog	33
Padrão de arquitetura hexagonal	34
Intenção	34
Motivação	34
Aplicabilidade	34
Problemas e considerações	35
Implementação	35
Arquitetura de alto nível	36
Implementação usando Serviços da AWS	37
Código de exemplo	38
Conteúdo relacionado	42
Vídeos	42
Padrão publicar/assinar	43
Intenção	43
Motivação	43
Aplicabilidade	43
Problemas e considerações	44
Implementação	45
Arquitetura de alto nível	45
Implementação usando serviços da AWS	46
Workshop	48
Referências do blog	48
Conteúdo relacionado	48
Tente novamente com o padrão de recuo	49

Intenção	49
Motivação	49
Aplicabilidade	49
Problemas e considerações	49
Implementação	50
Arquitetura de alto nível	50
Implementação usando serviços AWS	51
Código de exemplo	52
Repositório GitHub	53
Conteúdo relacionado	53
Padrões saga	54
Coreografia saga	55
Orquestração da saga	55
Coreografia saga	56
Intenção	56
Motivação	57
Aplicabilidade	57
Problemas e considerações	58
Implementação	59
Conteúdo relacionado	62
Orquestração da saga	62
Intenção	62
Motivação	62
Aplicabilidade	63
Problemas e considerações	63
Implementação	64
Referências do blog	69
Conteúdo relacionado	69
Vídeos	69
Padrão scatter-gather	70
Intenção	70
Motivação	70
Aplicabilidade	70
Problemas e considerações	71
Implementação	72
Arquitetura de alto nível	72

Implementação usando Serviços da AWS	75
Workshop	78
Referências do blog	78
Conteúdo relacionado	78
padrão strangler fig	79
Intenção	79
Motivação	79
Aplicabilidade	80
Problemas e considerações	80
Implementação	82
Arquitetura de alto nível	82
Implementação usando AWS serviços	87
Workshop	91
Referências do blog	91
Conteúdo relacionado	91
Padrão de caixa de saída transacional	92
Intenção	92
Motivação	92
Aplicabilidade	92
Problemas e considerações	93
Implementação	93
Arquitetura de alto nível	93
Implementação usando AWS serviços	94
Código de exemplo	99
Uso de uma tabela de outbox	99
Uso da captura de dados de alteração (CDC)	100
GitHub repositório	102
Recursos	103
Histórico do documento	104
Glossário	106
#	106
A	107
B	110
C	112
D	115
E	119

F	121
G	123
H	124
eu	126
L	128
M	129
O	134
P	136
Q	139
R	140
S	143
T	147
U	148
V	149
W	149
Z	150
.....	clii

Padrões, arquiteturas e implementações de design de nuvem

Anitha Deenadayalan, Amazon Web Services (AWS)

Maio de 2024 ([histórico do documento](#))

Este guia fornece orientação para implementar padrões de design de modernização comumente usados usando serviços AWS. Um número cada vez maior de aplicativos modernos é projetado usando arquiteturas de microsserviços para obter escalabilidade, melhorar a velocidade de lançamento, reduzir o escopo do impacto das mudanças e reduzir a regressão. Isso leva a uma maior produtividade do desenvolvedor e maior agilidade, melhor inovação e maior foco nas necessidades comerciais. As arquiteturas de microsserviços também apoiam o uso da melhor tecnologia para o serviço e o banco de dados e promovem o código poliglota e a persistência poliglota.

Tradicionalmente, os aplicativos monolíticos são executados em um único processo, usam um armazenamento de dados e são executados em servidores que escalam verticalmente. Em comparação, os aplicativos de microsserviços modernos são refinados, têm domínios de falha independentes, são executados como serviços em toda a rede e podem usar mais de um armazenamento de dados, dependendo do caso de uso. Os serviços são escalados horizontalmente e uma única transação pode abranger vários bancos de dados. As equipes de desenvolvimento devem concentrar-se na comunicação de rede, na persistência poliglota, na escalabilidade horizontal, na consistência eventual e no tratamento de transações nos armazenamentos de dados ao desenvolver aplicativos usando arquiteturas de microsserviços. Portanto, os padrões de modernização são essenciais para resolver problemas comuns no desenvolvimento de aplicativos modernos e ajudam a acelerar a entrega de software.

Este guia fornece uma referência técnica para arquitetos de nuvem, líderes técnicos, proprietários de aplicativos e negócios e desenvolvedores que desejam escolher a arquitetura de nuvem certa para padrões de design com base nas melhores práticas bem arquitetadas. Cada padrão discutido neste guia aborda um ou mais cenários conhecidos em arquiteturas de microsserviços. O guia discute os problemas e considerações associados a cada padrão, fornece uma implementação arquitetônica de alto nível e descreve a implementação do padrão na AWS. Exemplos de código aberto do GitHub e links de workshops são fornecidos quando disponíveis.

O guia aborda os seguintes padrões:

- [Camada anticorrupção](#)
- [Padrões de roteamento da API:](#)
 - [Roteamento de nome de host](#)
 - [Roteamento de caminho](#)
 - [Roteamento de cabeçalho HTTP](#)
- [Disjuntor](#)
- [Fornecimento de eventos](#)
- [Arquitetura hexagonal](#)
- [Publicar/assinar](#)
- [Tente novamente com o recuo](#)
- [Padrões saga:](#)
 - [Coreografia saga](#)
 - [Orquestração saga](#)
- [Scatter-gather](#)
- [Strangler fig](#)
- [Caixa de saída transacional](#)

Resultados de negócios desejados

Usando os padrões discutidos neste guia para modernizar os seus aplicativos, você pode:

- Projete e implemente arquiteturas confiáveis, seguras e operacionalmente eficientes, otimizadas para custo e desempenho.
- Reduza o tempo de ciclo para casos de uso que exigem esses padrões, para que você possa se concentrar nos desafios específicos da organização.
- Acelere o desenvolvimento padronizando implementações de padrões usando os serviços da AWS.
- Ajude seus desenvolvedores a criar aplicativos modernos sem herdar dívidas técnicas.

Padrão da camada anticorrupção

Intenção

O padrão de camada anticorrupção (ACL) atua como uma camada de mediação que converte a semântica do modelo de domínio de um sistema para outro. Ele converte o modelo do contexto limitado upstream (monólito) em um modelo adequado ao contexto limitado downstream (microserviço) antes de consumir o contrato de comunicação estabelecido pela equipe ascendente. Esse padrão pode ser aplicável quando o contexto limitado downstream contém um subdomínio principal ou o modelo upstream é um sistema legado não modificável. Também reduz o risco de transformação e a interrupção dos negócios, evitando alterações nos chamadores quando suas chamadas precisam ser redirecionadas de forma transparente para o sistema de destino.

Motivação

Durante o processo de migração, quando uma aplicação monolítica é migrada para microserviços, pode haver mudanças na semântica do modelo de domínio do serviço recém-migrado. Quando os recursos do monólito são necessários para chamar esses microserviços, as chamadas devem ser roteadas para o serviço migrado sem exigir nenhuma alteração nos serviços de chamada. O padrão da ACL permite que o monólito chame os microserviços de forma transparente, atuando como um adaptador ou uma camada de fachada que converte as chamadas para a semântica mais recente.

Aplicabilidade

Considere usar esse padrão quando:

- Sua aplicação monolítica existente precisa se comunicar com uma função que foi migrada para um microserviço, e o modelo e a semântica do domínio de serviço migrado diferem do recurso original.
- Dois sistemas têm semânticas diferentes e precisam trocar dados, mas não é prático modificar um sistema para ser compatível com o outro sistema.
- Você quer usar uma abordagem rápida e simplificada para adaptar um sistema a outro com o mínimo impacto.
- Sua aplicação está se comunicando com um sistema externo.

Problemas e considerações

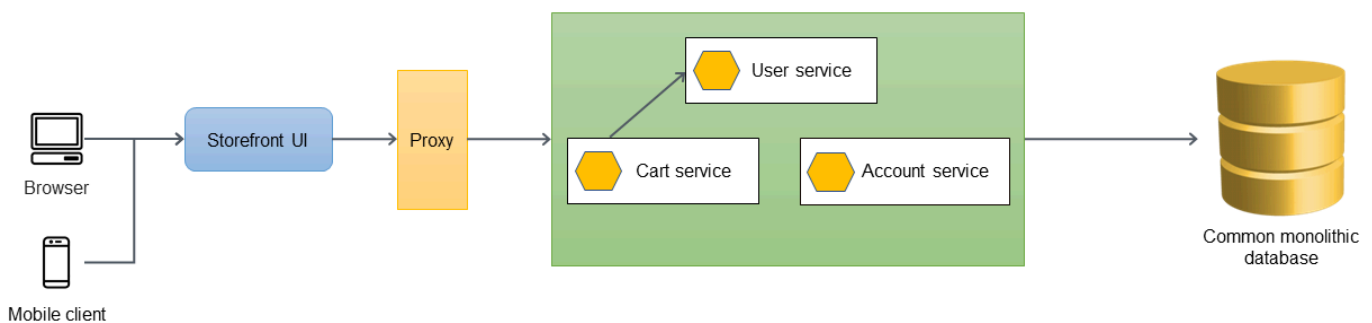
- Dependências da equipe: quando diferentes serviços em um sistema pertencem a equipes diferentes, a nova semântica do modelo de domínio nos serviços migrados pode levar a mudanças nos sistemas de chamada. No entanto, as equipes podem não conseguir fazer essas mudanças de forma coordenada, pois podem ter outras prioridades. A ACL separa os receptores e converte as chamadas para corresponder à semântica dos novos serviços, evitando assim a necessidade de os chamadores fazerem alterações no sistema atual.
- Sobrecarga operacional: o padrão da ACL exige esforço adicional para operar e manter. Esse trabalho inclui a integração da ACL com ferramentas de monitoramento e alerta, o processo de lançamento e os processos de integração e entrega contínuas (CI/CD).
- Ponto único de falha: qualquer falha na ACL pode tornar o serviço de destino inacessível, causando problemas na aplicação. Para mitigar esse problema, você deve incorporar recursos de repetição e disjuntores. Consulte a nova [tentativa com os padrões de recuo](#) e [disjuntor](#) para entender mais sobre essas opções. A configuração de alertas e registros apropriados melhorará o tempo médio de resolução (MTTR).
- Dívida técnica: como parte de sua estratégia de migração ou modernização, considere se a ACL será uma solução transitória ou provisória ou uma solução de longo prazo. Se for uma solução provisória, você deve registrar a ACL como uma dívida técnica e desativá-la depois que todos os chamadores dependentes tiverem sido migrados.
- Latência: a camada adicional pode introduzir latência devido à conversão de solicitações de uma interface para outra. Recomendamos que você defina e teste a tolerância de performance em aplicações sensíveis ao tempo de resposta antes de implantar a ACL em ambientes de produção.
- Gargalo de escalabilidade: em aplicações de alta carga em que os serviços podem ser escalados até o pico de carga, a ACL pode se tornar um gargalo e causar problemas de escalabilidade. Se o serviço de destino for escalado sob demanda, você deverá projetar a ACL para escalar adequadamente.
- Implementação compartilhada ou específica do serviço: você pode criar a ACL como um objeto compartilhado para converter e redirecionar chamadas para vários serviços ou classes específicas de serviços. Leve em consideração a latência, a escalabilidade e a tolerância a falhas ao determinar o tipo de implementação da ACL.

Implementação

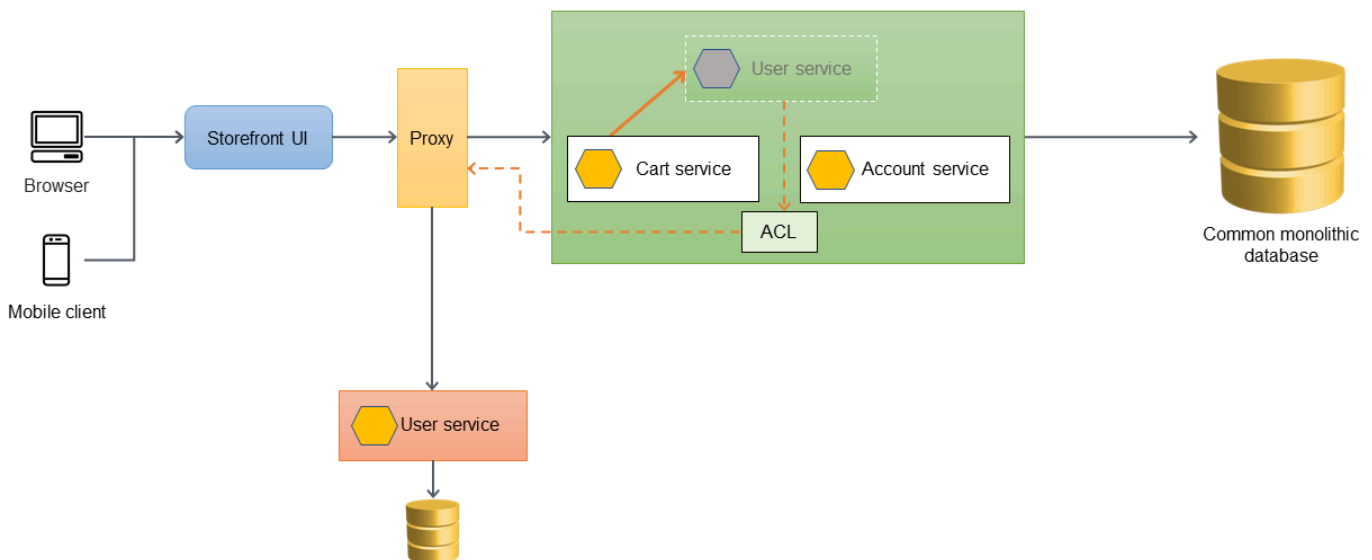
Você pode implementar a ACL dentro da sua aplicação monolítica como uma classe específica para o serviço que está sendo migrado ou como um serviço independente. A ACL deve ser desativada após a migração de todos os serviços dependentes para a arquitetura de microsserviços.

Arquitetura de alto nível

No exemplo de arquitetura a seguir, uma aplicação monolítica tem três serviços: serviço de usuário, serviço de carrinho e serviço de conta. O serviço de carrinho depende do serviço do usuário, e a aplicação usa um banco de dados relacional monolítico.

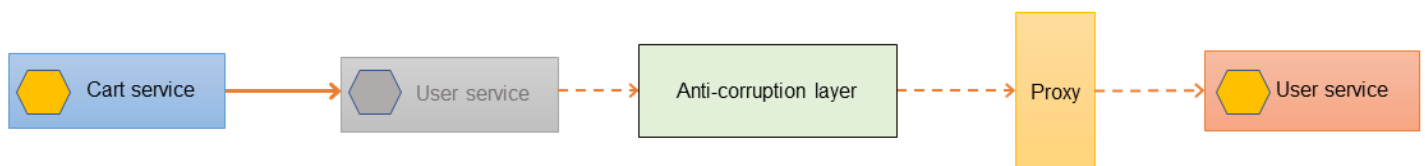


Na arquitetura a seguir, o serviço do usuário foi migrado para um novo microsserviço. O serviço de carrinho chama o serviço de usuário, mas a implementação não está mais disponível no monólito. Também é provável que a interface do serviço recém-migrado não corresponda à interface anterior, quando estava dentro da aplicação monolítica.



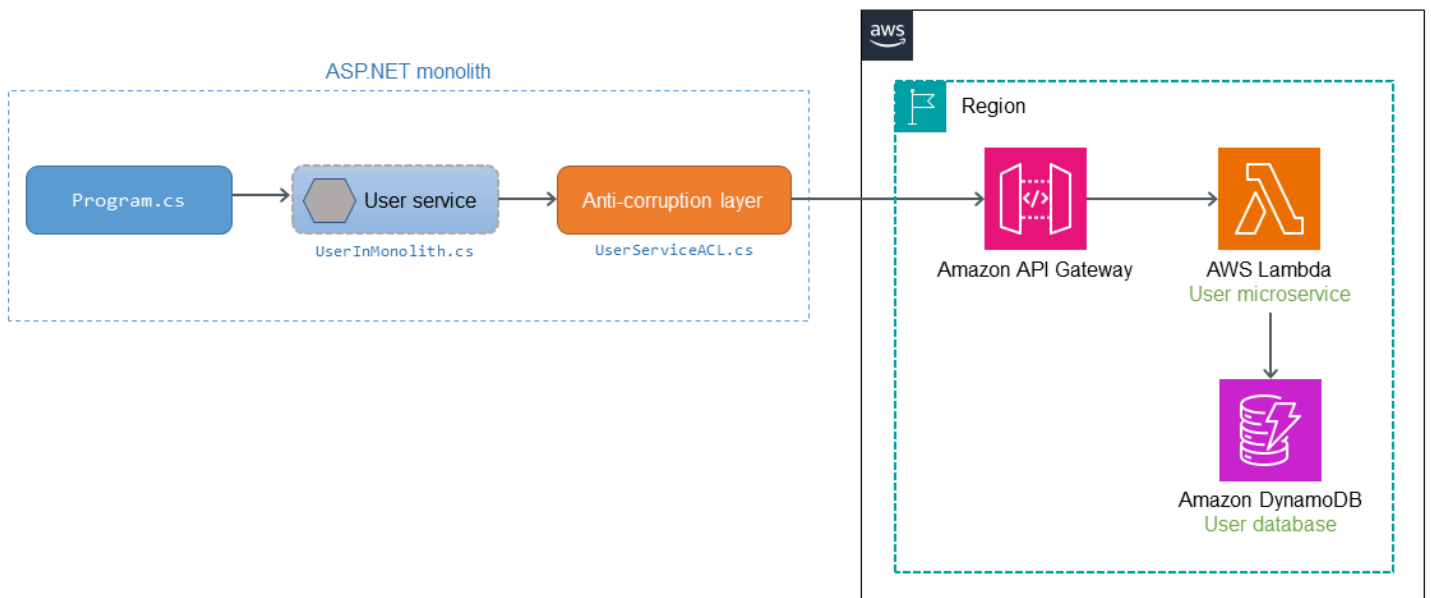
Se o serviço de carrinho precisar ligar diretamente para o serviço de usuário recém-migrado, isso exigirá alterações no serviço de carrinho e um teste completo da aplicação monolítica. Isso pode aumentar o risco de transformação e a interrupção dos negócios. O objetivo deve ser minimizar as alterações na funcionalidade existente da aplicação monolítica.

Nesse caso, recomendamos que você introduza uma ACL entre o serviço de usuário antigo e o serviço de usuário recém-migrado. A ACL funciona como um adaptador ou uma fachada que converte as chamadas na interface mais recente. A ACL pode ser implementada dentro da aplicação monolítica como uma classe (por exemplo, `UserServiceFacade` ou `UserServiceAdapter`) específica do serviço que foi migrado. A camada anticorrupção deverá ser desativada depois que todos os serviços dependentes tiverem sido migrados para a arquitetura de microsserviços.



Implementação usando serviços AWS

O diagrama a seguir mostra como você pode implementar esse exemplo de ACL usando serviços da AWS.



O microsserviço do usuário é migrado da aplicação ASP.NET monolítica e implantado como uma função do [AWS Lambda](#) na AWS. As chamadas para a função do Lambda são roteadas pelo

[Amazon API Gateway](#). A ACL é implantada no monólito para converter a chamada e se adaptar à semântica do microserviço do usuário.

Quando `Program.cs` chama o serviço do usuário (`UserInMonolith.cs`) dentro do monólito, a chamada é roteada para a ACL (`UserServiceACL.cs`). A ACL converte a chamada para a nova semântica e interface e chama o microserviço por meio do endpoint do API Gateway. O chamador (`Program.cs`) não está ciente da conversão e do roteamento que ocorrem no serviço ao usuário e na ACL. Como o chamador não está ciente das mudanças no código, há menos interrupções nos negócios e menor risco de transformação.

Código de exemplo

O trecho de código a seguir fornece as alterações no serviço original e a implementação de `UserServiceACL.cs`. Quando uma solicitação é recebida, o serviço de usuário original chama a ACL. A ACL converte o objeto de origem para corresponder à interface do serviço recém-migrado, chama o serviço e retorna a resposta ao chamador.

```
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
MediaTyewithQualityHeaderValue("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;
    }
}
```

Repositório GitHub

Para uma implementação completa da arquitetura de amostra desse padrão, consulte o repositório do GitHub em <https://github.com/aws-samples/anti-corruption-layer-pattern>.

Conteúdo relacionado

- [padrão strangler fig](#)
- [Padrão de disjuntor](#)
- [Tente novamente com o padrão de recuo](#)

Padrões de roteamento de API

Em ambientes de desenvolvimento ágil, equipes autônomas (por ex., esquadrões e tribos) possuem um ou mais serviços que incluem muitos microsserviços. As equipes expõem esses serviços como APIs para permitir que seus consumidores interajam com seu grupo de serviços e ações.

Há três métodos principais para expor as APIs HTTP aos consumidores upstream usando nomes de host e caminhos:

Method (Método)	Descrição	Exemplo
Roteamento de nomes de host	Exponha cada serviço como um nome de host.	<code>billing.api.example.com</code>
Roteamento de caminhos	Exponha cada serviço como um caminho.	<code>api.example.com/billing</code>
Roteamento com base no cabeçalho	Exponha cada serviço como um cabeçalho HTTP.	<code>x-example-action: something</code>

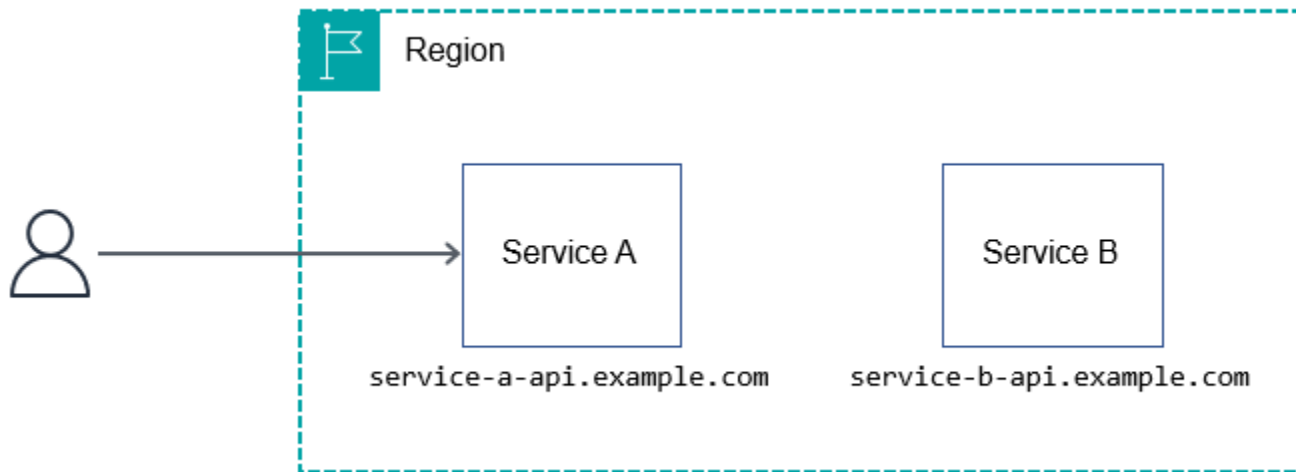
Esta seção descreve os casos de uso típicos desses três métodos de roteamento e suas vantagens e desvantagens para ajudá-lo a decidir qual método se adapta melhor aos seus requisitos e estrutura organizacional.

Padrão de roteamento por nome de host

O roteamento por nome de host é um mecanismo para isolar serviços de API, dando a cada API seu próprio nome de host; por exemplo, `service-a.api.example.com` ou `service-a.example.com`.

Caso de uso típico

O roteamento usando nomes de host reduz a quantidade de atrito nas versões, pois nada é compartilhado entre as equipes de serviço. As equipes são responsáveis por gerenciar tudo, desde entradas de DNS até operações de serviço em produção.



Prós

O roteamento de nome de host é, de longe, o método mais simples e escalável para roteamento de API HTTP. É possível usar qualquer serviço AWS relevante para criar uma arquitetura que siga esse método – você pode criar uma arquitetura com [Amazon API Gateway](#), [AWS AppSync](#), [Application Load Balancers](#) e [Amazon Elastic Compute Cloud \(Amazon EC2\)](#) ou qualquer outro serviço compatível com HTTP.

As equipes podem usar o roteamento de nome de host para possuir totalmente seu subdomínio. Também fica mais fácil isolar, testar e orquestrar implantações para Regiões da AWS específicos ou versões; por exemplo, `region.service-a.api.example.com` ou `dev.region.service-a.api.example.com`.

Contras

Quando você usa o roteamento de nome de host, seus consumidores precisam se lembrar de nomes de host diferentes para interagir com cada API que você expõe. Você pode mitigar esse problema fornecendo um SDK de cliente. No entanto, os SDKs do cliente vêm com seu próprio conjunto de desafios. Por exemplo, eles precisam oferecer suporte a atualizações contínuas, vários idiomas, versionamento, comunicação de alterações significativas causadas por problemas de segurança ou correções de bugs, documentação e assim por diante.

Ao usar o roteamento de nome de host, você também precisa registrar o subdomínio ou domínio toda vez que criar um novo serviço.

Padrão de roteamento de caminhos

O roteamento por caminhos é o mecanismo de agrupar várias ou todas as APIs sob o mesmo nome de host e usar um URI de solicitação para isolar serviços; por exemplo, `api.example.com/service-a` ou `api.example.com/service-b`.

Caso de uso típico

A maioria das equipes opta por esse método porque deseja uma arquitetura simples — um desenvolvedor precisa se lembrar de apenas uma URL, como `api.example.com`, para interagir com a API HTTP. A documentação da API geralmente é mais fácil de digerir porque geralmente é mantida em conjunto em vez de ser dividida em diferentes portais ou PDFs.

O roteamento com base no caminho é considerado um mecanismo simples para compartilhar uma API HTTP. No entanto, envolve sobrecarga operacional, como configuração, autorização, integrações e latência adicional devido a vários saltos. Também requer processos maduros de gerenciamento de mudanças para garantir que uma configuração incorreta não interrompa todos os serviços.

Em AWS, há várias maneiras de compartilhar uma API e rotear de forma eficaz para o serviço correto. As seções a seguir discutem três abordagens: proxy reverso de serviço HTTP, API Gateway e Amazon CloudFront. Nenhuma das abordagens sugeridas para unificar os serviços de API depende dos serviços posteriores em execução no AWS. Os serviços podem ser executados em qualquer lugar sem problemas ou com qualquer tecnologia, desde que sejam compatíveis com HTTP.

Proxy reverso de serviço HTTP

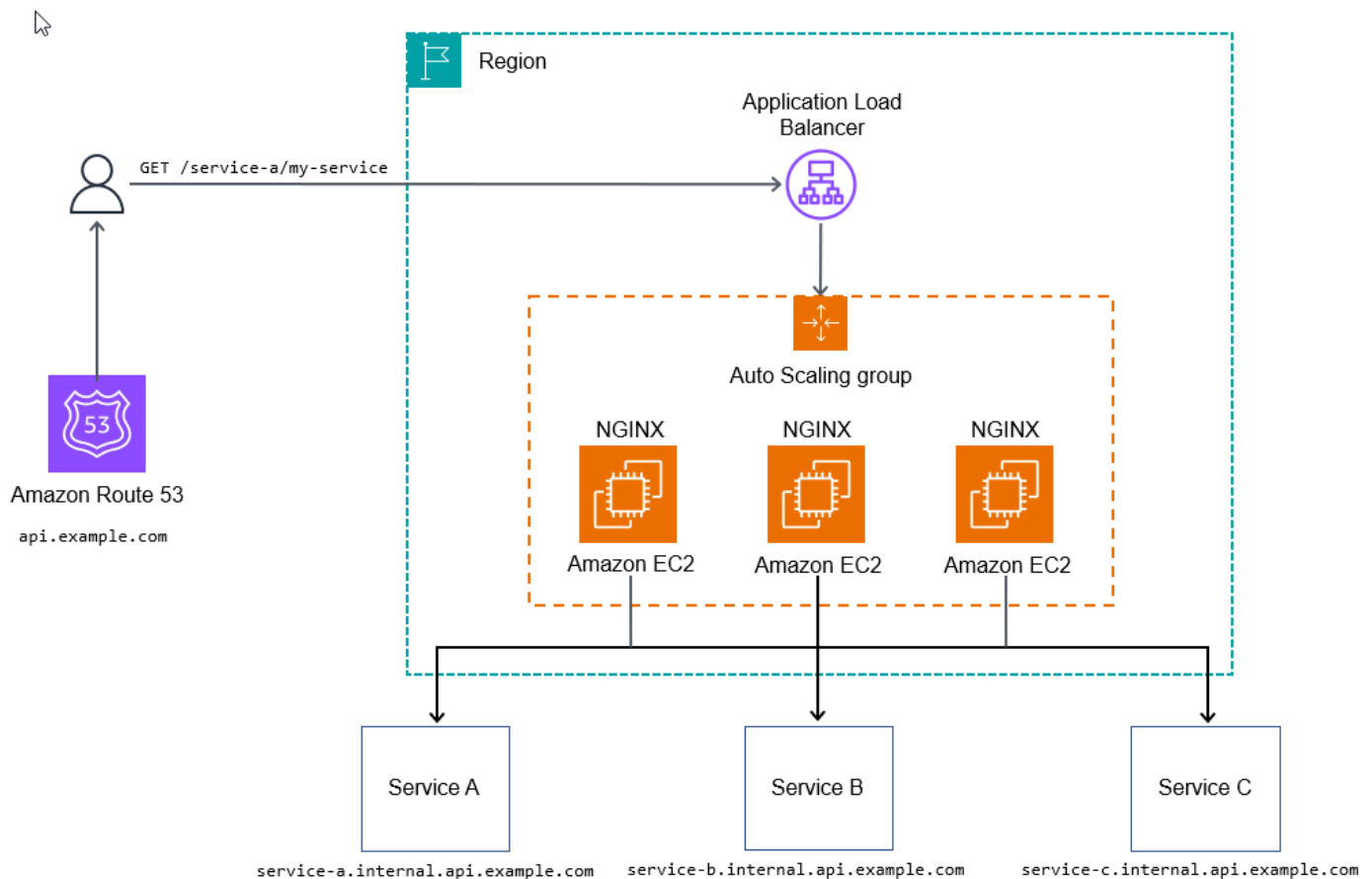
Você pode usar um servidor HTTP, como o [NGINX](#), para criar configurações de roteamento dinâmico. Em uma arquitetura [Kubernetes](#), você também pode criar uma regra de entrada para corresponder a um caminho para um serviço. (Este guia não aborda a entrada do Kubernetes; consulte a [documentação do Kubernetes](#) para obter mais informações.)

A configuração a seguir para o NGINX mapeia dinamicamente uma solicitação HTTP de `api.example.com/my-service/` para `my-service.internal.api.example.com`.

```
server {  
    listen 80;
```

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

O diagrama a seguir ilustra o método de proxy reverso de serviço HTTP.



Essa abordagem pode ser suficiente para alguns casos de uso que não usam configurações adicionais para iniciar o processamento de solicitações, permitindo que a API posterior colete métricas e registros.

Para se preparar para a prontidão de produção operacional, você deve poder adicionar observabilidade para todos os níveis de sua pilha, adicionar configurações adicionais ou adicionar scripts para personalizar seu ponto de entrada de API para permitir atributos mais avançados, como limitação de taxa ou tokens de uso.

Prós

O objetivo final do método de proxy reverso de serviço HTTP é criar uma abordagem escalável e gerenciável para unificar APIs em um único domínio, de forma que pareça coerente para qualquer consumidor de API. Essa abordagem também permite que suas equipes de serviço implantem e gerenciem suas próprias APIs, com o mínimo de sobrecarga após a implantação. Serviços gerenciados AWS para rastreamento, como [AWS X-Ray](#) ou [AWS WAF](#), ainda são aplicáveis aqui.

Contras

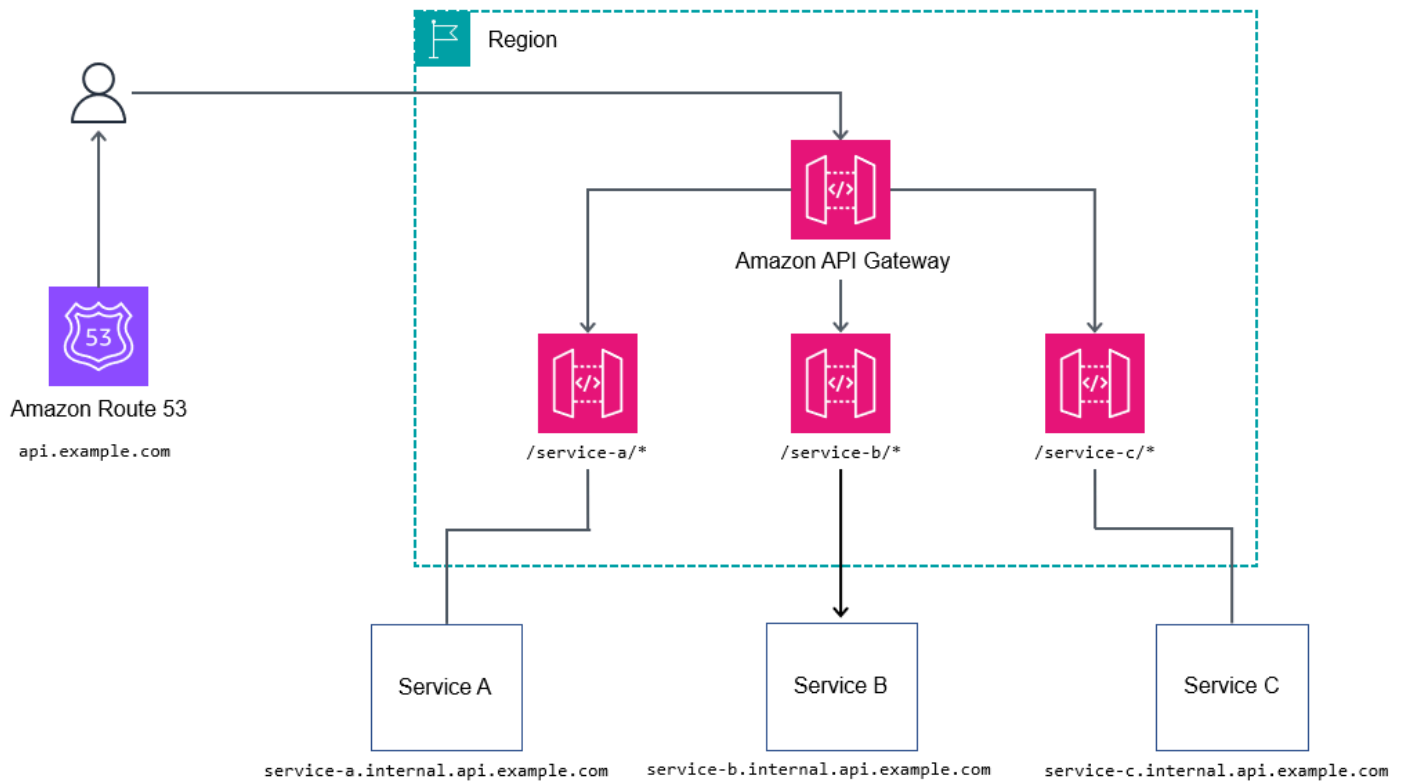
A principal desvantagem dessa abordagem são os testes e o gerenciamento extensivos dos componentes de infraestrutura necessários, embora isso possa não ser um problema se você tiver equipes de engenharia de confiabilidade de sites (SRE) instaladas.

Há um ponto de inflexão de custos com esse método. Em volumes baixos a médios, ele é mais caro do que alguns dos outros métodos discutidos neste guia. Em grandes volumes, é muito econômico (cerca de 100 mil transações por segundo ou mais).

API Gateway

O serviço [Amazon API Gateway](#) (APIs REST e APIs HTTP) pode rotear o tráfego de forma semelhante ao método de proxy reverso do serviço HTTP. Usar um gateway de API no modo proxy HTTP fornece uma maneira simples de agrupar muitos serviços em um ponto de entrada para o subdomínio de nível superior `api.example.com` e, em seguida, solicitações de proxy para o serviço aninhado; por exemplo, `billing.internal.api.example.com`.

Provavelmente não é uma boa ideia ser muito granular mapeando todos os caminhos em todos os serviços no gateway de API raiz ou principal. Em vez disso, opte por caminhos curinga, como o `/billing/*`, para encaminhar solicitações para o serviço de cobrança. Ao não mapear todos os caminhos no gateway de API raiz ou principal, você ganha mais flexibilidade em relação às suas APIs, porque não precisa atualizar o gateway de API raiz a cada alteração da API.



Prós

Para controlar fluxos de trabalho mais complexos, como alterar os atributos da solicitação, as APIs REST expõem a Linguagem de Modelo Apache Velocity (VTL) para permitir que você modifique a solicitação e a resposta. As APIs REST podem fornecer benefícios adicionais, como estes:

- [Aut N/Z com AWS Identity and Access Management \(IAM\), Amazon Cognito ou autorizadores AWS Lambda](#)
- [AWS X-Ray para rastreamento](#)
- [Integração com AWS WAF](#)
- [Limitação de taxa básica](#)
- Tokens de uso para agrupar consumidores em diferentes níveis (consulte [Acelerar solicitações de API para obter melhor throughput](#) na documentação do API Gateway)

Contras

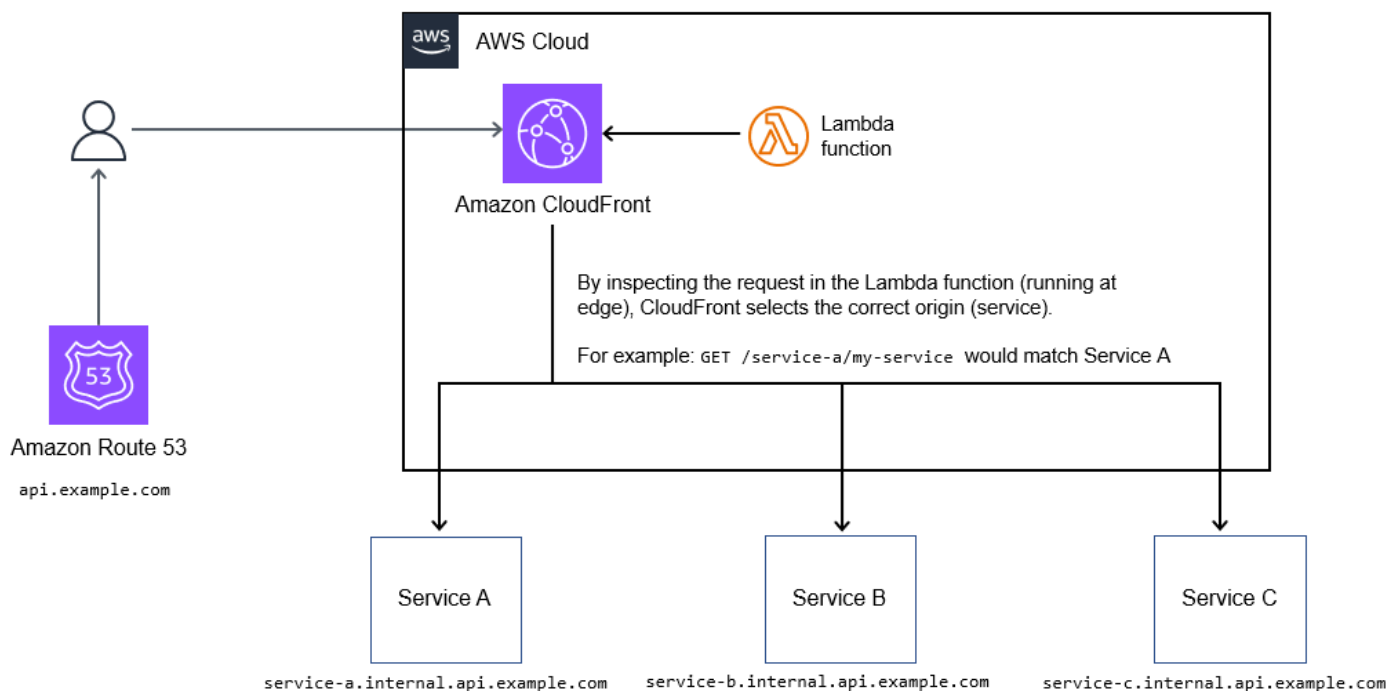
Em grandes volumes, o custo pode ser um problema para alguns usuários.

CloudFront

Você pode usar o [atributo de seleção dinâmica de origem](#) no [Amazon CloudFront](#) para selecionar condicionalmente uma origem (um serviço) para encaminhar a solicitação. Você pode usar esse atributo para rotear vários serviços por meio de um único nome de host, como `api.example.com`.

Caso de uso típico

A lógica de roteamento vive como código dentro da função Lambda@Edge, portanto, ela oferece suporte a mecanismos de roteamento altamente personalizáveis, como testes A/B, lançamentos canário, sinalização de atributo e reescrita de caminhos. Isso é ilustrado no diagrama a seguir.



Prós

Se você precisa armazenar em cache as respostas da API, esse método é uma boa maneira de unificar uma coleção de serviços em um único endpoint. É um método econômico para unificar coleções de APIs.

Além disso, o CloudFront oferece suporte à [criptografia em nível de campo](#), bem como à integração com o AWS WAF para limitação básica de taxa e ACLs básicas.

Contras

Esse método suporta no máximo 250 origens (serviços) que podem ser unificadas. Esse limite é suficiente para a maioria das implantações, mas pode causar problemas com um grande número de APIs à medida que você aumenta seu portfólio de serviços.

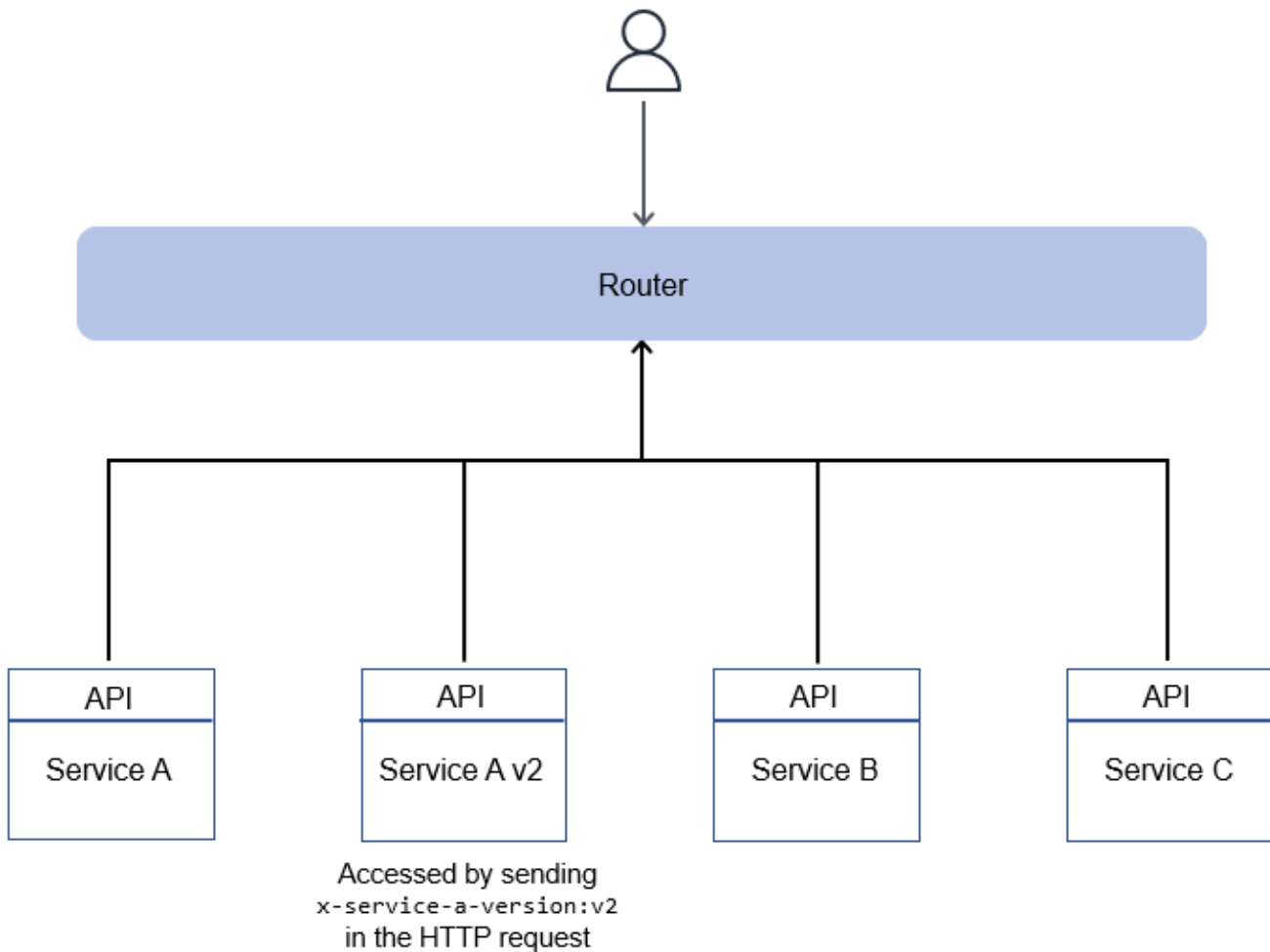
Atualmente, a atualização das funções do Lambda@Edge leva alguns minutos. O CloudFront também leva até 30 minutos para concluir a propagação das alterações em todos os pontos de presença. Em última análise, isso bloqueia novas atualizações até que elas sejam concluídas.

Padrão de roteamento de cabeçalho HTTP

O roteamento com base no cabeçalho permite direcionar o serviço correto para cada solicitação especificando um cabeçalho HTTP na solicitação HTTP. Por exemplo, enviar o cabeçalho `x-service-a-action: get-thing` permitiria `get thing` a partir de Service A. O caminho da solicitação ainda é importante, pois oferece orientação sobre em qual atributo você está tentando trabalhar.

Além de usar o roteamento de cabeçalho HTTP para ações, você pode usá-lo como um mecanismo para roteamento de versões, habilitando sinalizadores de atributos, testes A/B ou necessidades semelhantes. Na realidade, você provavelmente usará o roteamento de cabeçalho com um dos outros métodos de roteamento para criar APIs robustas.

A arquitetura do roteamento de cabeçalho HTTP normalmente tem uma fina camada de roteamento na frente dos microsserviços que encaminha para o serviço correto e retorna uma resposta, conforme ilustrado no diagrama a seguir. Essa camada de roteamento pode cobrir todos os serviços ou apenas alguns serviços para habilitar uma operação, como roteamento com base na versão.



Prós

As alterações de configuração exigem o mínimo esforço e podem ser automatizadas facilmente. Esse método também é flexível e oferece suporte a formas criativas de expor somente operações específicas que você desejaria de um serviço.

Contras

Assim como no método de roteamento de nome de host, o roteamento de cabeçalho HTTP pressupõe que você tenha controle total sobre o cliente e possa manipular cabeçalhos HTTP personalizados. Proxies, redes de entrega de conteúdo (CDNs) e balanceadores de carga podem limitar o tamanho do cabeçalho. Embora seja improvável que isso seja uma preocupação, pode ser um problema dependendo de quantos cabeçalhos e cookies você adiciona.

Padrão de disjuntor

Intenção

O padrão do disjuntor pode impedir que um serviço de chamada repita uma chamada para outro serviço (receptor) quando a chamada já gerou repetidos tempos limite ou falhas. O padrão também é usado para detectar quando o serviço do receptor está funcional novamente.

Motivação

Quando vários microsserviços colaboram para lidar com solicitações, um ou mais serviços podem ficar indisponíveis ou apresentar alta latência. Quando aplicações complexas usam microsserviços, uma interrupção em um microsserviço pode levar à falha da aplicação. Os microsserviços se comunicam por meio de chamadas de procedimentos remotos, e erros transitórios podem ocorrer na conectividade da rede, causando falhas. (Os erros transitórios podem ser tratados usando o padrão de [repetição com recuo](#).) Durante a execução síncrona, a cascata de tempos limite ou falhas pode causar uma experiência ruim para o usuário.

No entanto, em algumas situações, as falhas podem levar mais tempo para serem resolvidas, por exemplo, quando o serviço do receptor está inativo ou uma contenção no banco de dados resulta em tempos limite. Nesses casos, se o serviço de chamada repetir as chamadas repetidamente, essas novas tentativas poderão resultar em contenção na rede e no consumo do grupo de threads do banco de dados. Além disso, se vários usuários tentarem a aplicação novamente e de forma repetida, isso piorará o problema e poderá causar degradação da performance de toda a aplicação.

O padrão do disjuntor foi popularizado por Michael Nygard em seu livro *Release It* (Nygard 2018). Esse padrão de design pode impedir que um serviço de chamada repita uma chamada de serviço que já tenha gerado repetidos tempos limite ou falhas. Ele também pode detectar quando o serviço do receptor está funcional novamente.

Os objetos do disjuntor funcionam como disjuntores elétricos que interrompem automaticamente a corrente quando há uma anormalidade no circuito. Os disjuntores elétricos desligam ou interrompem o fluxo da corrente quando há uma falha. Da mesma forma, o objeto do disjuntor está situado entre o chamador e o serviço do receptor e dispara se o receptor não estiver disponível.

As [falácias da computação distribuída](#) são um conjunto de afirmações feitas por Peter Deutsch e outros da Sun Microsystems. Eles dizem que programadores que são novos em aplicações

distribuídas invariavelmente fazem suposições falsas. A confiabilidade da rede, as expectativas de latência zero e as limitações de largura de banda resultam em aplicações de software escritas com o mínimo de tratamento de erros de rede.

Durante uma interrupção na rede, as aplicações podem esperar por uma resposta indefinidamente e consumir continuamente os recursos da aplicação. Deixar de repetir as operações quando a rede estiver disponível também pode levar à degradação da aplicação. Se as chamadas de API para um banco de dados ou um serviço externo expirarem devido a problemas de rede, chamadas repetidas sem disjuntor podem afetar o custo e a performance.

Aplicabilidade

Use esse padrão quando:

- O serviço de chamadas faz uma chamada que provavelmente falhará.
- A alta latência exibida pelo serviço do receptor (por exemplo, quando as conexões do banco de dados são lentas) gera tempos limite no serviço do chamador.
- O serviço do chamador faz uma chamada síncrona, mas o serviço do receptor não está disponível ou apresenta alta latência.

Problemas e considerações

- Implementação independente de serviço: para evitar o excesso de código, recomendamos que você implemente o objeto disjuntor de forma independente de microsserviços e orientada por API.
- Fechamento do circuito pelo receptor: quando o receptor se recupera de um problema ou falha de performance, ele pode atualizar o status do circuito para CLOSED. Esta é uma extensão do padrão do disjuntor e poderá ser implementada se seu objetivo de tempo de recuperação (RTO) exigir.
- Chamadas multithreaded: o valor do tempo limite de expiração é definido como o período de tempo em que o circuito permanece desligado antes que as chamadas sejam roteadas novamente para verificar a disponibilidade do serviço. Quando o serviço do receptor é chamado em vários threads, a primeira chamada que falhou define o valor do tempo limite de expiração. Sua implementação deve garantir que as chamadas subsequentes não alterem o tempo limite de expiração indefinidamente.
- Forçar a abertura ou o fechamento do circuito: os administradores do sistema devem ter a capacidade de abrir ou fechar um circuito. Isso pode ser feito atualizando o valor do tempo limite de expiração na tabela do banco de dados.

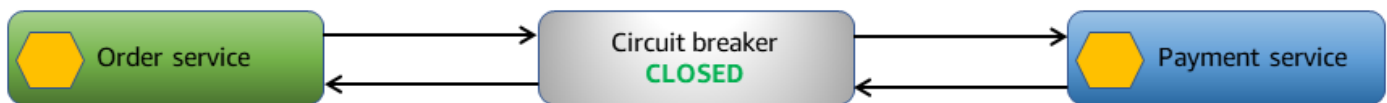
- Observabilidade: a aplicação deve ter o registro em log configurado para identificar as chamadas que falham quando o disjuntor está aberto.

Implementação

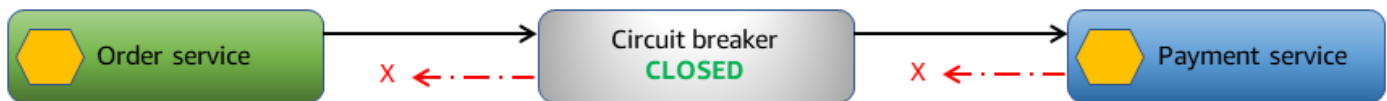
Arquitetura de alto nível

No exemplo a seguir, o chamador é o serviço de pedidos e o receptor é o serviço de pagamento.

Quando não há falhas, o serviço de pedidos roteia todas as chamadas para o serviço de pagamento pelo disjuntor, conforme mostra o diagrama a seguir.



Se o serviço de pagamento atingir o tempo limite, o disjuntor poderá detectar o tempo limite e rastrear a falha.



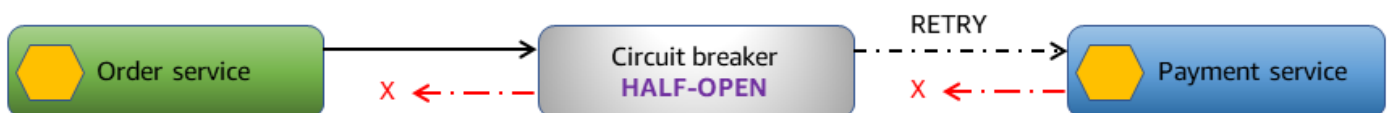
Circuit breaker with payment service failure

Se os tempos limite excederem um limite especificado, a aplicação abrirá o circuito. Quando o circuito está aberto, o objeto do disjuntor não roteia as chamadas para o serviço de pagamento. Ele retorna uma falha imediata quando o serviço de pedidos chama o serviço de pagamento.



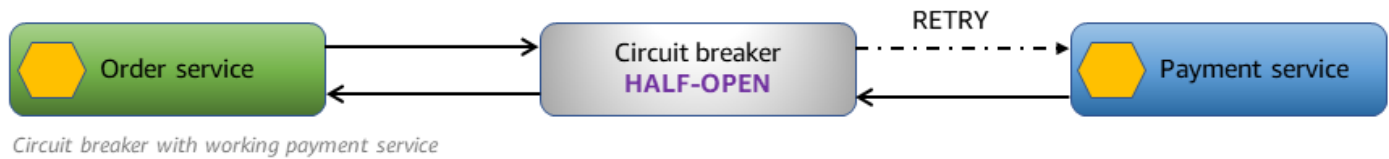
Circuit breaker stops routing to payment service

O objeto do disjuntor tenta periodicamente verificar se as chamadas para o serviço de pagamento foram bem-sucedidas.



Circuit breaker periodically retries payment service

Quando a chamada para o serviço de pagamento é bem-sucedida, o circuito é fechado e todas as chamadas adicionais são roteadas para o serviço de pagamento novamente.



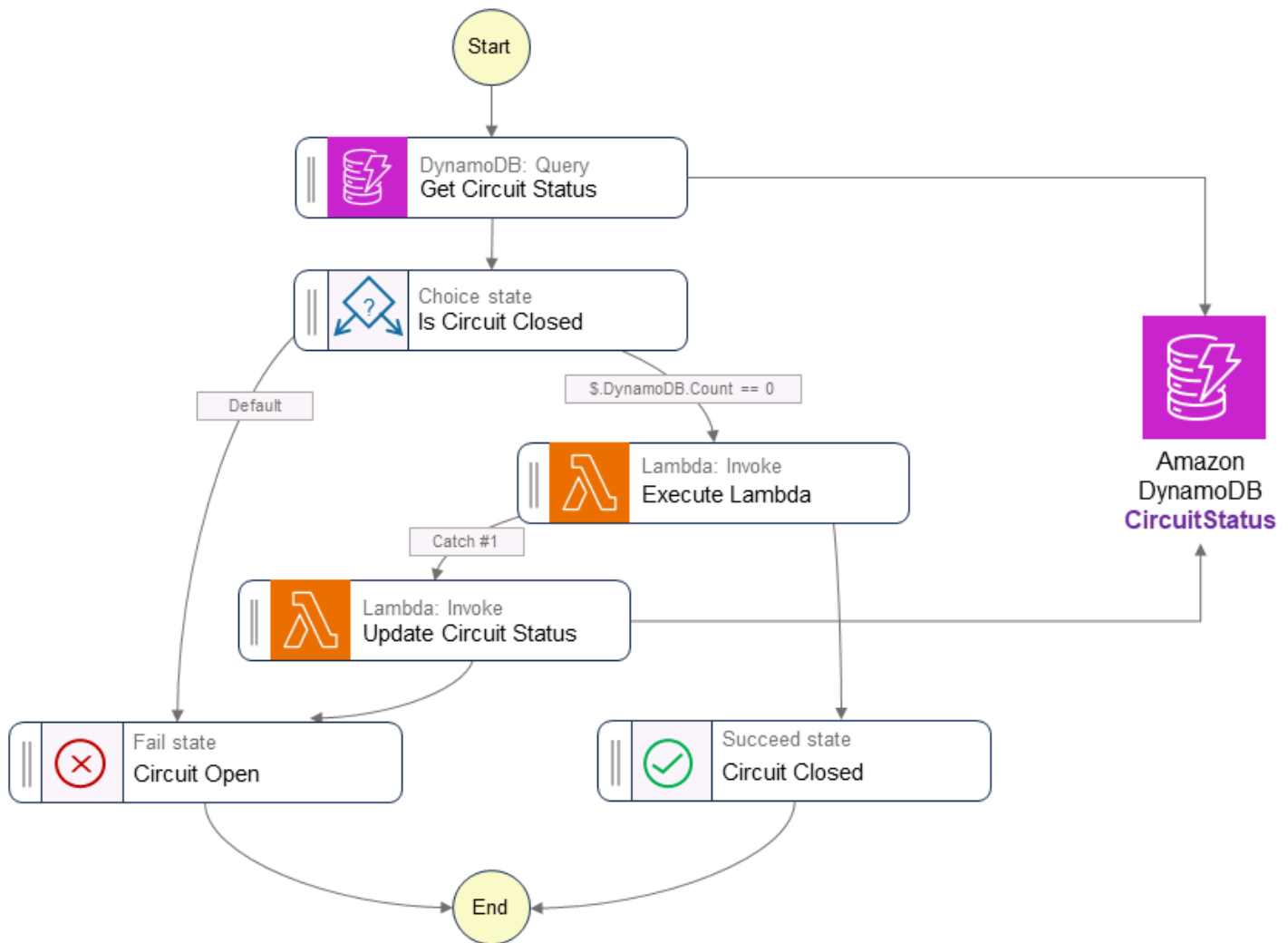
Implementação usando AWS serviços

A solução de amostra usa fluxos de trabalho expressos no [AWS Step Functions](#) para implementar o padrão do disjuntor. A máquina de estado do Step Functions permite que você configure os recursos de repetição e o fluxo de controle baseado em decisões necessários para a implementação do padrão.

A solução também usa uma tabela do [Amazon DynamoDB](#) como armazenamento de dados para rastrear o status do circuito. Isso pode ser substituído por um armazenamento de dados na memória, como o [Amazon ElastiCache \(Redis OSS\)](#), para melhor desempenho.

Quando um serviço deseja chamar outro serviço, ele inicia o fluxo de trabalho com o nome do serviço do receptor. O fluxo de trabalho obtém o status do disjuntor na tabela `CircuitStatus` do DynamoDB, que armazena os serviços atualmente degradados. Se `CircuitStatus` contiver um registro não expirado para o receptor, o circuito estará aberto. O fluxo de trabalho do Step Functions retorna uma falha imediata e tem como saída um estado FAIL.

Se a tabela `CircuitStatus` não contiver um registro para o receptor ou contiver um registro expirado, o serviço estará operacional. A etapa `ExecuteLambda` na definição da máquina de estado chama a função do Lambda que é enviada por meio de um valor de parâmetro. Se a chamada for bem-sucedida, o fluxo de trabalho do Step Functions será encerrado com um estado SUCCESS.



Se a chamada de serviço falhar ou ocorrer um tempo limite, a aplicação tentará novamente com um recuo exponencial por um número definido de vezes. Se a chamada de serviço falhar após as novas tentativas, o fluxo de trabalho vai inserir um registro na tabela `CircuitStatus` do serviço com um `ExpiryTimeStamp`, e o fluxo de trabalho sairá com um estado `FAIL`. As chamadas subsequentes para o mesmo serviço retornarão uma falha imediata, desde que o disjuntor esteja aberto. A etapa `Get Circuit Status` na definição da máquina de estado verifica a disponibilidade do serviço com base no valor `ExpiryTimeStamp`. Os itens expirados são excluídos da tabela `CircuitStatus` usando o recurso de tempo de vida (TTL) do DynamoDB.

Código de exemplo

O código a seguir usa a função `GetCircuitStatus` do Lambda para verificar o status do disjuntor.

```
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 = "";
}
```

O código a seguir mostra as instruções do Amazon States Language no fluxo de trabalho do Step Functions.

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

GitHub repositório

Para uma implementação completa da arquitetura de amostra desse padrão, consulte o GitHub repositório em <https://github.com/aws-samples/circuit-breaker-netcore-blog>.

Referências do blog

- [Usando o padrão de disjuntor com o Amazon AWS Step Functions DynamoDB](#)

Conteúdo relacionado

- [Padrão strangler fig](#)
- [Tente novamente com o padrão de recuo](#)

Padrão de fornecimento de eventos

Intenção

Em arquiteturas orientadas a eventos, o padrão de fornecimento de eventos armazena os eventos que resultam em uma mudança de estado em um armazenamento de dados. Isso ajuda a capturar e manter um histórico completo das mudanças de estado e promove auditabilidade, rastreabilidade e a capacidade de analisar estados anteriores.

Motivação

Vários microsserviços podem colaborar para lidar com solicitações e se comunicam por meio de eventos. Esses eventos podem resultar em uma mudança no estado (dados). Armazenar objetos de eventos na ordem em que eles ocorrem fornece informações valiosas sobre o estado atual da entidade de dados e informações adicionais sobre como ela chegou a esse estado.

Aplicabilidade

Use o padrão de fornecimento de eventos quando:

- É necessário um histórico imutável dos eventos que ocorrem em um aplicativo para o rastreamento.
- Projeções de dados políglotas são exigidas de uma única fonte de verdade (SSOT).
- É necessária a reconstrução pontual do estado do aplicativo.
- O armazenamento a longo prazo do estado do aplicativo não é necessário, mas talvez você queira reconstruí-lo conforme necessário.
- As workloads têm volumes diferentes de leitura e gravação. Por exemplo, você tem workloads com muita gravação que não exigem processamento em tempo real.
- A captura de dados de alteração (CDC) é necessária para analisar o desempenho do aplicativo e outras métricas.
- Os dados de auditoria são necessários para todos os eventos que acontecem em um sistema para fins de emissão de relatórios e conformidade.
- Você deseja derivar cenários hipotéticos alterando (inserindo, atualizando ou excluindo) eventos durante o processo de repetição para determinar o possível estado final.

Problemas e considerações

- **Controle otimista de concorrência:** esse padrão armazena todos os eventos que causam uma mudança de estado no sistema. Vários usuários ou serviços podem tentar atualizar os mesmos dados ao mesmo tempo, causando colisões de eventos. Essas colisões acontecem quando eventos conflitantes são criados e aplicados ao mesmo tempo, o que resulta em um estado final de dados que não corresponde à realidade. Para resolver esse problema, você pode implementar estratégias para detectar e resolver colisões de eventos. Por exemplo, você pode implementar um esquema otimista de controle de concorrência incluindo versionamento ou adicionando registros de data e hora a eventos para rastrear a ordem das atualizações.
- **Complexidade:** a implementação do fornecimento de eventos exige uma mudança de mentalidade das operações tradicionais de CRUD para o pensamento orientado por eventos. O processo de repetição, usado para restaurar o sistema ao estado original, pode ser complexo para garantir a idempotência dos dados. Repositório de eventos, backups e instantâneos também podem aumentar a complexidade.
- **Consistência eventual:** as projeções de dados derivadas dos eventos acabam sendo consistentes devido à latência na atualização de dados usando o padrão de segmentação de responsabilidade de consulta de comando (CQRS) ou visões materializadas. Quando os consumidores processam dados de um repositório de eventos e os publicadores enviam novos dados, a projeção de dados ou o objeto do aplicativo podem não representar o estado atual.
- **Consulta:** a recuperação de dados atuais ou agregados dos logs de eventos pode ser mais complexa e lenta em comparação com bancos de dados tradicionais, especialmente para consultas complexas e tarefas de geração de relatórios. Para mitigar esse problema, o fornecimento de eventos geralmente é implementado com o padrão CQRS.
- **Tamanho e custo do repositório de eventos:** o repositório de eventos pode experimentar um crescimento exponencial em tamanho à medida que os eventos persistem, especialmente em sistemas com alta taxa de throughput de eventos ou períodos de retenção prolongados. Conseqüentemente, você deve arquivar periodicamente os dados do evento em um armazenamento econômico para evitar que o repositório de eventos fique muito grande.
- **Escalabilidade do repositório de eventos:** o repositório de eventos deve lidar de forma eficiente com grandes volumes de operações de gravação e leitura. Escalar um repositório de eventos pode ser um desafio, por isso é importante ter um repositório de dados que forneça fragmentos e partições.
- **Eficiência e otimização:** escolha ou projete um repositório de eventos que gerencie as operações de gravação e leitura com eficiência. O repositório de eventos deve ser otimizado para o

volume de eventos e os padrões de consulta esperados para o aplicativo. A implementação de mecanismos de indexação e consulta pode acelerar a recuperação de eventos ao reconstruir o estado do aplicativo. Você também pode considerar o uso de bancos de dados ou bibliotecas especializadas em repositórios de eventos que ofereçam atributos de otimização de consultas.

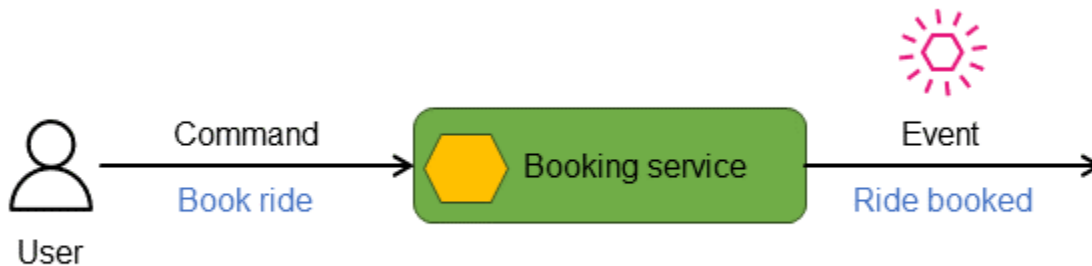
- **Snapshots:** você deve fazer backup dos logs de eventos em intervalos regulares com ativação baseada no tempo. A repetição dos eventos no último backup bem-sucedido conhecido dos dados deve levar à recuperação pontual do estado do aplicativo. O objetivo de ponto de recuperação (RPO) é o tempo máximo aceitável desde o último ponto de recuperação de dados. O RPO determina o que é considerado uma perda aceitável de dados entre o último ponto de recuperação e a interrupção do serviço. A frequência dos instantâneos diários do repositório de dados e eventos deve ser baseada no RPO do aplicativo.
- **Sensibilidade temporal:** os eventos são armazenados na ordem em que ocorrem. Portanto, a confiabilidade da rede é um fator importante a ser considerado ao implementar esse padrão. Problemas de latência podem levar a um estado incorreto do sistema. Use filas de primeiro a entrar, primeiro a sair (FIFO) com entrega no máximo uma vez para levar os eventos até o repositório do evento.
- **Desempenho de repetição de eventos:** pode ser demorado repetir um número substancial de eventos para reconstruir o estado atual do aplicativo. São necessários esforços de otimização para melhorar o desempenho, principalmente ao reproduzir eventos de dados arquivados.
- **Atualizações externas do sistema:** os aplicativos que usam o padrão de fornecimento de eventos podem atualizar os armazenamentos de dados em sistemas externos e podem capturar essas atualizações como objetos de eventos. Durante as repetições de eventos, isso pode se tornar um problema se o sistema externo não esperar uma atualização. Nesses casos, você pode usar sinalizadores de atributos para controlar as atualizações externas do sistema.
- **Consultas do sistema externo:** quando as chamadas externas do sistema são confidenciais à data e hora da chamada, os dados recebidos podem ser armazenados em armazenamentos de dados internos para uso durante as repetições.
- **Versionamento de eventos:** à medida que o aplicativo evolui, a estrutura dos eventos (esquema) pode mudar. É necessário implementar uma estratégia de versionamento para eventos para garantir a compatibilidade com versões anteriores e futuras. Isso pode envolver a inclusão de um campo de versão na carga útil do evento e o tratamento adequado de diferentes versões do evento durante a repetição.

Implementação

Arquitetura de alto nível

Comandos e eventos

Em aplicativos de microsserviços distribuídos e orientados por eventos, os comandos representam as instruções ou solicitações enviadas a um serviço, normalmente com a intenção de iniciar uma mudança em seu estado. O serviço processa esses comandos e avalia a validade e a aplicabilidade do comando em seu estado atual. Se o comando for executado com êxito, o serviço responderá emitindo um evento que indica a ação tomada e as informações de estado relevantes. Por exemplo, no diagrama a seguir, o serviço de reserva responde ao comando Reservar corrida emitindo o evento Corrida reservada.



Repositórios de eventos

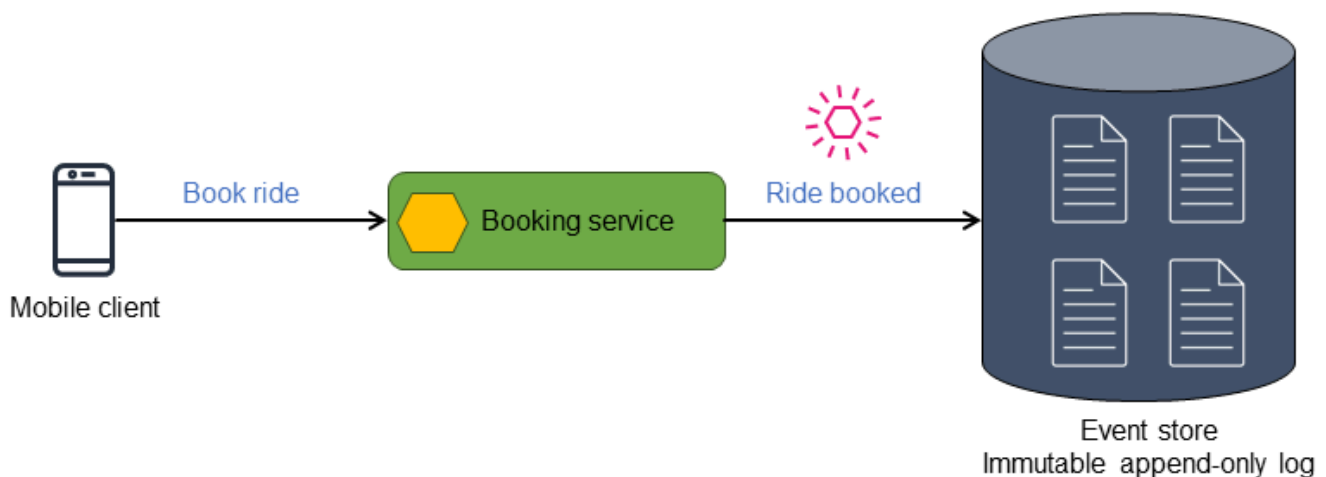
Os eventos são registrados em um armazenamento de dados ou repositório imutável, somente para anexos e ordenado cronologicamente, conhecido como repositório de eventos. Cada mudança de estado é tratada como um objeto de evento individual. Um objeto de entidade ou um repositório de dados com um estado inicial conhecido, seu estado atual e qualquer visualização pontual pode ser reconstruído reproduzindo os eventos na ordem de sua ocorrência.

O repositório de eventos atua como um registro histórico de todas as ações e mudanças de estado e serve como uma valiosa fonte única de verdade. Você pode usar o repositório de eventos para derivar o estado final e atualizado do sistema passando os eventos por um processador de repetição, que aplica esses eventos para produzir uma representação precisa do estado mais recente do sistema. Você também pode usar o repositório de eventos para gerar a perspectiva pontual do estado reproduzindo os eventos por meio de um processador de repetição. No padrão de

fornecimento de eventos, o estado atual pode não ser totalmente representado pelo objeto de evento mais recente. Você pode derivar o estado atual de três maneiras:

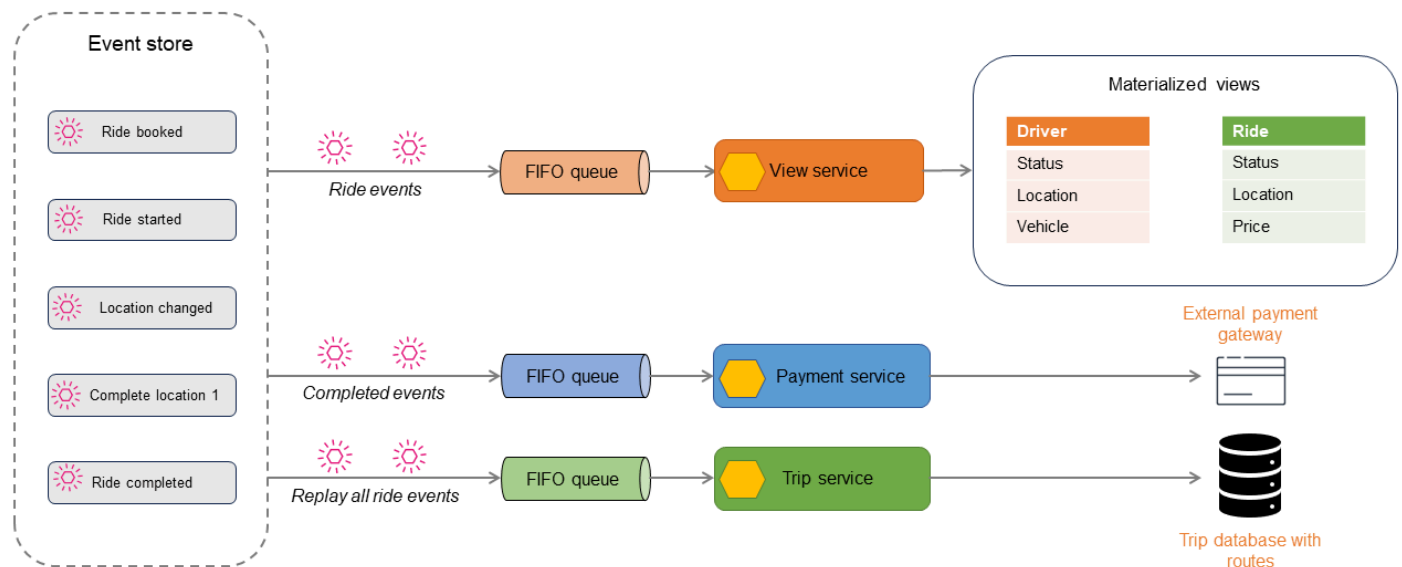
- Ao agregar eventos relacionados. Os objetos de eventos relacionados são combinados para gerar o estado atual para consulta. Essa abordagem geralmente é usada em conjunto com o padrão CQRS, em que os eventos são combinados e gravados no repositório de dados somente para leitura.
- Usando visões materializadas. Você pode empregar o fornecimento de eventos com o padrão de visão materializada para calcular ou resumir os dados do evento e obter o estado atual dos dados relacionados.
- Reproduzindo eventos. Objetos de eventos podem ser reproduzidos para realizar ações para gerar o estado atual.

O diagrama a seguir mostra o evento Ride booked sendo armazenado em um repositório de eventos.



O repositório de eventos publica os eventos que armazena, e os eventos podem ser filtrados e roteados para o processador apropriado para ações subsequentes. Por exemplo, os eventos podem ser roteados para um processador de visualização que resume o estado e mostra uma visão materializada. Os eventos são transformados no formato de dados do repositório de dados de destino. Essa arquitetura pode ser estendida para derivar diferentes tipos de repositórios de dados, o que leva à persistência poliglota dos dados.

O diagrama a seguir descreve os eventos em um aplicativo de reserva de corrida. Todos os eventos que ocorrem no aplicativo são armazenados no repositório de eventos. Os eventos armazenados são então filtrados e encaminhados para diferentes consumidores.



Os eventos de corrida podem ser usados para gerar repositórios de dados somente para leitura usando o CQRS ou o padrão de visão materializada. Você pode obter o estado atual da corrida, do motorista ou da reserva consultando os repositórios de leitura. Alguns eventos, como *Location changed* ou *Ride completed*, são publicados para outro consumidor para processamento de pagamentos. Quando a corrida é concluída, todos os eventos da corrida são repetidos para criar um histórico da corrida para fins de auditoria ou geração de relatórios.

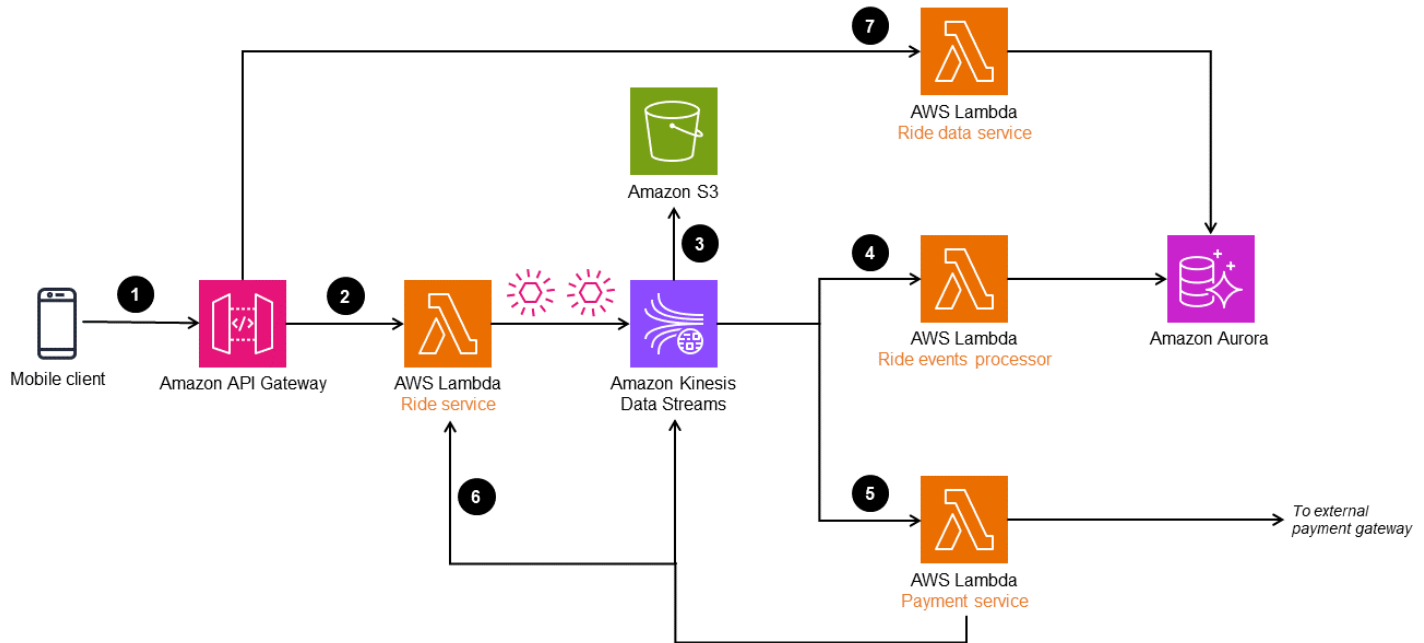
O padrão de fornecimento de eventos é frequentemente usado em aplicativos que exigem uma recuperação pontual e também quando os dados precisam ser projetados em formatos diferentes usando uma única fonte confiável. Ambas as operações exigem um processo de repetição para executar os eventos e derivar o estado final necessário. O processador de repetição também pode exigir um ponto de partida conhecido — de preferência não a partir da abertura do aplicativo, pois esse não seria um processo eficiente. Recomendamos que você tire instantâneos regulares do estado do sistema e aplique um número menor de eventos para obter um estado atualizado.

Implementação usando serviços da AWS

Na arquitetura a seguir, o Amazon Kinesis Data Streams é usado como repositório de eventos. Esse serviço captura e gerencia as alterações do aplicativo como eventos e oferece uma solução de fluxo de dados em tempo real e de alto throughput. Para implementar o padrão de fornecimento

de eventos na AWS, você também pode usar serviços como o Amazon EventBridge e o Amazon Managed Streaming for Apache Kafka (Amazon MSK) com base nas necessidades do seu aplicativo.

Para aumentar a durabilidade e habilitar a auditoria, você pode arquivar os eventos que são capturados pelo Kinesis Data Streams no Amazon Simple Storage Service (Amazon S3). Essa abordagem de armazenamento duplo ajuda a reter dados históricos de eventos com segurança para fins futuros de análise e conformidade.



O fluxo de trabalho consiste no seguinte:

1. Uma solicitação de reserva de corrida é feita por meio de um cliente móvel para um endpoint do Amazon API Gateway.
2. O microsserviço de corrida (função `Ride service` do Lambda) recebe a solicitação, transforma os objetos e publica no Kinesis Data Streams.
3. Os dados do evento no Kinesis Data Streams são armazenados no Amazon S3 para fins de conformidade e histórico de auditoria.
4. Os eventos são transformados e processados pela função `Ride event processor` do Lambda e armazenados em um banco de dados Amazon Aurora para fornecer uma visão materializada dos dados da corrida.
5. Os eventos de corrida concluídos são filtrados e enviados para processamento de pagamento em um gateway de pagamento externo. Quando o pagamento é concluído, outro evento é enviado ao Kinesis Data Streams para atualizar o banco de dados da Corrida.

6. Quando a corrida é concluída, os eventos da corrida são reproduzidos na função `Ride service` do Lambda para criar rotas e o histórico da corrida.
7. As informações da corrida podem ser lidas por meio do `Ride data service`, que lê a partir do banco de dados Aurora.

O API Gateway também pode enviar o objeto do evento diretamente para o Kinesis Data Streams sem a função `Ride service` do Lambda. No entanto, em um sistema complexo, como um serviço de carona, o objeto do evento pode precisar ser processado e enriquecido antes de ser ingerido no fluxo de dados. Por esse motivo, a arquitetura tem um `Ride service` que processa o evento antes de enviá-lo para o Kinesis Data Streams.

Referências do blog

- [Novidade no AWS Lambda — FIFO do SQS como uma origem de evento](#)

Padrão de arquitetura hexagonal

Intenção

O padrão de arquitetura hexagonal, também conhecido como padrão de portas e adaptadores, foi proposto pelo Dr. Alistair Cockburn em 2005. O objetivo é criar arquiteturas fracamente acopladas nas quais os componentes do aplicativo possam ser testados de forma independente, sem dependências de armazenamentos de dados ou interfaces de usuário (). UIs Esse padrão ajuda a evitar o bloqueio tecnológico de armazenamentos de dados e. UIs Isso facilita a alteração da pilha de tecnologia ao longo do tempo, com impacto limitado ou inexistente na lógica de negócios. Nessa arquitetura com acoplamento fraco, a aplicação se comunica com componentes externos por meio de interfaces chamadas portas e usa adaptadores para converter as trocas técnicas com esses componentes.

Motivação

O padrão de arquitetura hexagonal é usado para isolar a lógica de negócios (lógica de domínio) do código de infraestrutura relacionado, como código para acessar um banco de dados ou externo. APIs Esse padrão é útil para criar lógica de negócios e código de infraestrutura fracamente acoplados para AWS Lambda funções que exigem integração com serviços externos. Nas arquiteturas tradicionais, uma prática comum é incorporar a lógica de negócios na camada do banco de dados como procedimentos armazenados e na interface do usuário. Essa prática, junto com o uso de constructos específicos de UI na lógica de negócios, leva a arquiteturas estritamente acopladas que causam gargalos nas migrações de bancos de dados e nos esforços de modernização da experiência do usuário (UX). O padrão de arquitetura hexagonal permite que você projete seus sistemas e aplicações por propósito e não por tecnologia. Essa estratégia resulta em componentes de aplicações facilmente intercambiáveis, como bancos de dados, UX e componentes de serviço.

Aplicabilidade

Use o padrão de arquitetura hexagonal quando:

- Você deseja desacoplar a arquitetura da sua aplicação para criar componentes que possam ser totalmente testados.
- Vários tipos de clientes podem usar a mesma lógica de domínio.

- Seus componentes de interface de usuário e banco de dados exigem atualizações periódicas de tecnologia que não afetem a lógica da aplicação.
- Sua aplicação requer vários provedores de entrada e consumidores de saída, e personalizar a lógica da aplicação leva à complexidade do código e à falta de extensibilidade.

Problemas e considerações

- Design orientado por domínio: a arquitetura hexagonal funciona especialmente bem com o design orientado por domínio (DDD). Cada componente da aplicação representa um subdomínio no DDD, e arquiteturas hexagonais podem ser usadas para obter um acoplamento fraco entre os componentes da aplicação.
- Testabilidade: por design, uma arquitetura hexagonal usa abstrações para entradas e saídas. Portanto, gravar testes de unidade e testar isoladamente torna-se mais fácil devido ao acoplamento fraco inerente.
- Complexidade: a complexidade de separar a lógica de negócios do código de infraestrutura, quando tratada com cuidado, pode trazer grandes benefícios, como agilidade, cobertura de testes e adaptabilidade tecnológica. Caso contrário, os problemas podem se tornar complexos de resolver.
- Sobrecarga de manutenção: o código adicional do adaptador que torna a arquitetura conectável é justificado somente se o componente da aplicação exigir várias fontes de entrada e destinos de saída para gravar, ou quando o armazenamento de dados de entrada e saída precisar mudar com o tempo. Caso contrário, o adaptador se tornará outra camada adicional a ser mantida, o que introduz uma sobrecarga de manutenção.
- Problemas de latência: o uso de portas e adaptadores adiciona outra camada, o que pode resultar em latência.

Implementação

As arquiteturas hexagonais oferecem suporte ao isolamento da lógica de aplicativos e negócios do código de infraestrutura e do código que integra o aplicativo a bancos de dados externos e APIs agentes de mensagens. Uls Você pode conectar facilmente os componentes da lógica de negócios a outros componentes (como bancos de dados) na arquitetura da aplicação por meio de portas e adaptadores.

As portas são pontos de entrada independentes de tecnologia em um componente da aplicação. Essas interfaces personalizadas determinam a interface que permite que atores externos se comuniquem com o componente da aplicação, independentemente de quem ou do que implementa a interface. Isso é semelhante à forma como uma porta USB permite que muitos tipos diferentes de dispositivos se comuniquem com um computador, desde que usem um adaptador USB.

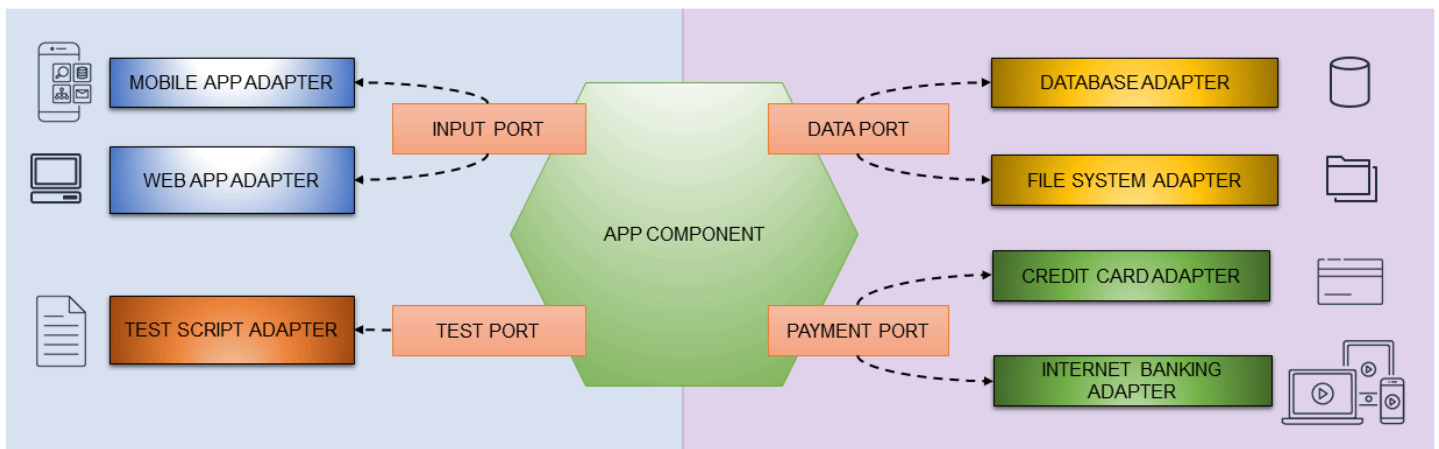
Os adaptadores interagem com a aplicação por meio de uma porta usando uma tecnologia específica. Os adaptadores se conectam a essas portas, recebem ou fornecem dados às portas e transformam os dados para processamento adicional. Por exemplo, um adaptador REST permite que os atores se comuniquem com o componente da aplicação por meio de uma API REST. Uma porta pode ter vários adaptadores sem nenhum risco para a porta ou para o componente da aplicação. Para estender o exemplo anterior, adicionar um adaptador GraphQL à mesma porta fornece um meio adicional para os atores interagirem com a aplicação por meio de uma API do GraphQL sem afetar a API REST, a porta ou a aplicação.

As portas se conectam à aplicação, e os adaptadores servem como uma conexão com o mundo externo. Você pode usar portas para criar componentes de aplicações com acoplamento fraco e trocar componentes dependentes mudando o adaptador. Isso permite que o componente da aplicação interaja com entradas e saídas externas sem precisar ter nenhum conhecimento contextual. Os componentes podem ser trocados em qualquer nível, o que facilita os testes automatizados. Você pode testar componentes de forma independente, sem dependências no código da infraestrutura, em vez de provisionar um ambiente inteiro para realizar testes. A lógica da aplicação não depende de fatores externos, então os testes são simplificados e fica mais fácil simular dependências.

Por exemplo, em uma arquitetura com acoplamento fraco, um componente da aplicação deve ser capaz de ler e gravar dados sem conhecer os detalhes do armazenamento de dados. A responsabilidade do componente da aplicação é fornecer dados para uma interface (porta). Um adaptador define a lógica de gravação em um armazenamento de dados, que pode ser um banco de dados, um sistema de arquivos ou um sistema de armazenamento de objetos, como o Amazon S3, dependendo das necessidades da aplicação.

Arquitetura de alto nível

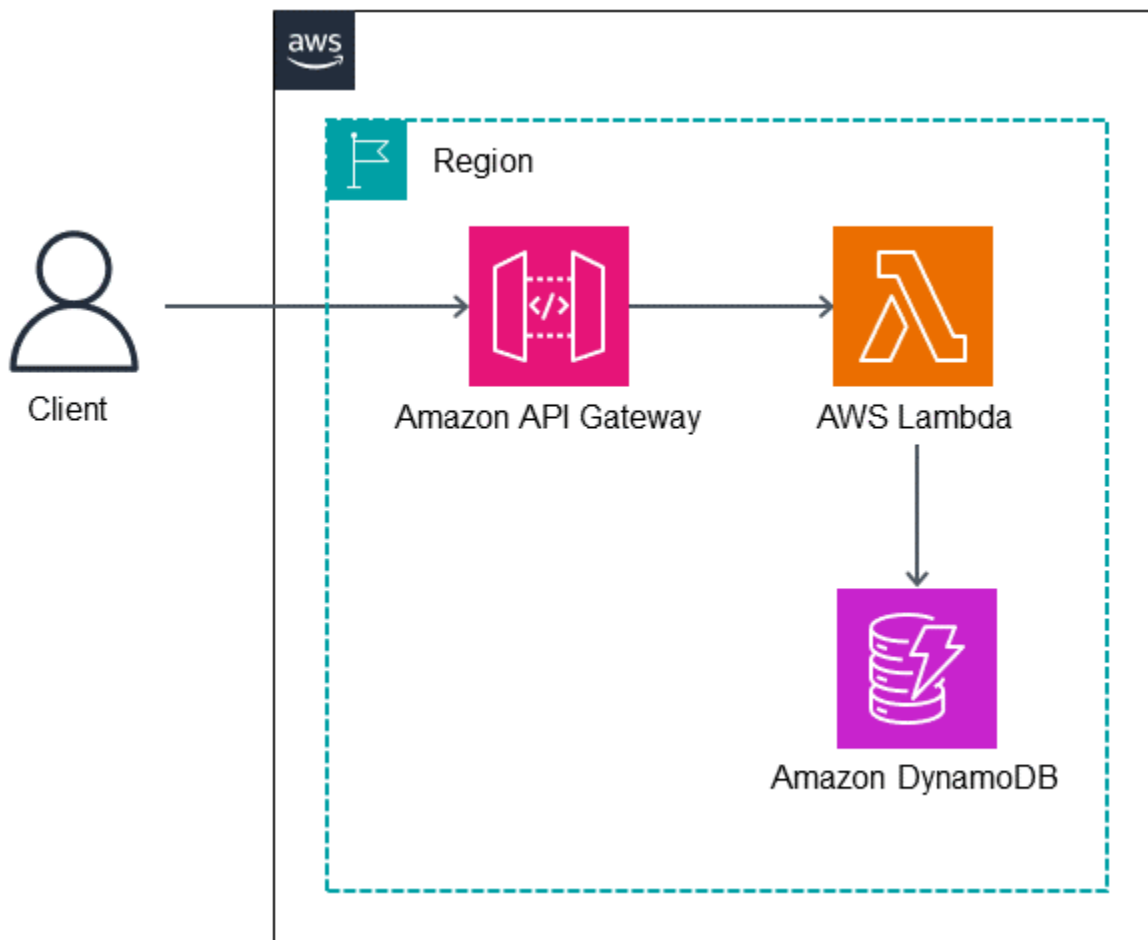
A aplicação ou o componente da aplicação contém a lógica de negócios principal. Ela recebe comandos ou consultas das portas e envia solicitações pelas portas para atores externos, que são implementadas por meio de adaptadores, conforme ilustrado no diagrama a seguir.



Implementação usando Serviços da AWS

AWS Lambda as funções geralmente contêm lógica de negócios e código de integração de banco de dados, que são fortemente acoplados para atingir um objetivo. Você pode usar o padrão de arquitetura hexagonal para separar a lógica de negócios do código de infraestrutura. Essa separação permite o teste de unidade da lógica de negócios sem dependências no código do banco de dados e melhora a agilidade do processo de desenvolvimento.

Na arquitetura a seguir, uma função do Lambda implementa o padrão de arquitetura hexagonal. A função do Lambda é iniciada pela API REST do Amazon API Gateway. A função implementa a lógica de negócios e grava dados nas tabelas do DynamoDB.



Código de exemplo

O código de exemplo nesta seção mostra como implementar o modelo de domínio usando o Lambda, como separá-lo do código de infraestrutura (como o código para acessar o DynamoDB) e como implementar testes de unidade para a função.

Modelo de domínio

A classe do modelo de domínio não tem conhecimento de componentes ou dependências externas, ela apenas implementa a lógica de negócios. No exemplo a seguir, a classe `Recipient` é uma classe de modelo de domínio que verifica se há sobreposições na data da reserva.

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

```

    self.__first_name = first_name
    self.__last_name = last_name
    self.__age = age
    self.__slots = []

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

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

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

```

Porta de entrada

A classe `RecipientInputPort` se conecta à classe do destinatário e executa a lógica do domínio.

```

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

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

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

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

```

```

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

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

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

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

```

Classe de adaptador do DynamoDB

A classe `DDBRecipientAdapter` implementa o acesso às tabelas do DynamoDB.

```

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

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

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

```

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

A função do Lambda `get_recipient_input_port` é uma fábrica para instâncias da classe `RecipientInputPort`. Ela cria instâncias de classes de portas de saída com instâncias de adaptador relacionadas.

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

def lambda_handler(event, context):

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

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

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

Teste de unidade

Você pode testar a lógica de negócios para classes de modelo de domínio injetando classes simuladas. O exemplo a seguir fornece o teste de unidade para a classe `Recipient` do modelo de domínio.

```
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 repositório

Para uma implementação completa da arquitetura de amostra desse padrão, consulte o GitHub repositório em <https://github.com/aws-samples/aws-lambda-domain-model-sample>.

Conteúdo relacionado

- [Hexagonal architecture](#), artigo de Alistair Cockburn
- [Desenvolvendo arquiteturas evolutivas com AWS Lambda](#) (postagem de AWS blog em japonês)

Vídeos

O vídeo a seguir (em japonês) analisa o uso da arquitetura hexagonal na implementação de um modelo de domínio usando uma função do Lambda.

Padrão publicar/assinar

Intenção

O padrão publicar/assinar, também conhecido como padrão pub-ass, é um padrão de mensagens que separa o remetente da mensagem (publicador) dos destinatários interessados (assinantes). Esse padrão implementa comunicações assíncronas publicando mensagens ou eventos por meio de um intermediário conhecido como agente de mensagens ou roteador (infraestrutura de mensagens). O padrão publicar/assinar aumenta a escalabilidade e a capacidade de resposta dos remetentes, transferindo a responsabilidade da entrega da mensagem para a infraestrutura da mensagem, para que o remetente possa se concentrar no processamento principal da mensagem.

Motivação

Em arquiteturas distribuídas, os componentes do sistema frequentemente precisam fornecer informações a outros componentes à medida que os eventos ocorrem dentro do sistema. O padrão publicar/assinar separa as preocupações para que os aplicativos possam se concentrar em seus recursos principais, enquanto a infraestrutura de mensagens lida com responsabilidades de comunicação, como roteamento de mensagens e entrega confiável. O padrão publicar/assinar permite que mensagens assíncronas dissociem o publicador e os assinantes. Os publicadores também podem enviar mensagens sem o conhecimento dos assinantes.

Aplicabilidade

Use o padrão publicar/assinar quando:

- O processamento paralelo é necessário se uma única mensagem tiver fluxos de trabalho diferentes.
- Não são necessárias a transmissão de mensagens para vários assinantes e respostas em tempo real dos destinatários.
- O sistema ou aplicativo pode tolerar uma consistência eventual dos dados ou do estado.
- O aplicativo ou componente precisa se comunicar com outros aplicativos ou serviços que possam usar linguagens, protocolos ou plataformas diferentes.

Problemas e considerações

- Disponibilidade do assinante: o publicador não sabe se os assinantes estão recebendo, e talvez não estejam. As mensagens publicadas são de natureza transitória e podem resultar no cancelamento se os assinantes não estiverem disponíveis.
- Garantia de entrega de mensagens: normalmente, o padrão publicar/assinar não pode garantir a entrega de mensagens para todos os tipos de assinantes, embora determinados serviços, como o Amazon Simple Notification Service (Amazon SNS), possam fornecer entrega [exatamente uma vez](#) para alguns subconjuntos de assinantes.
- Tempo de vida (TTL): as mensagens têm vida útil e expiram se não forem processadas dentro desse período. Considere adicionar as mensagens publicadas a uma fila para que elas possam persistir e garantir o processamento além do período TTL.
- Relevância da mensagem: os produtores podem definir um intervalo de tempo para a relevância como parte dos dados da mensagem, e a mensagem pode ser descartada após essa data. Considere fazer com que os consumidores examinem essas informações antes de decidir como processar a mensagem.
- Consistência eventual: há um atraso entre o momento em que a mensagem é publicada e o momento em que é consumida pelo assinante. Isso pode fazer com que os armazenamentos de dados do assinante se tornem eventualmente consistentes quando for necessária uma consistência forte. A consistência eventual também pode ser um problema quando produtores e consumidores precisam de interação quase em tempo real.
- Comunicação unidirecional: o padrão publicar/assinar é considerado unidirecional. Os aplicativos que exigem mensagens bidirecionais com um canal de assinatura de retorno devem considerar o uso de um padrão de solicitação-resposta se for necessária uma resposta síncrona.
- Ordem das mensagens: a ordem das mensagens não é garantida. Se os consumidores precisarem de mensagens solicitadas, recomendamos que você use os [tópicos FIFO do Amazon SNS](#) para garantir o pedido.
- Duplicação de mensagens: com base na infraestrutura de mensagens, mensagens duplicadas podem ser entregues aos consumidores. Os consumidores devem ser projetados para serem idempotentes para lidar com o processamento de mensagens duplicadas. Como alternativa, use os [tópicos FIFO do Amazon SNS](#) para garantir uma entrega exata uma vez.
- Filtragem de mensagens: os consumidores geralmente estão interessados apenas em um subconjunto das mensagens publicadas por um publicador. Forneça mecanismos para permitir que os assinantes filtrem ou restrinjam as mensagens que recebem fornecendo tópicos ou filtros de conteúdo.

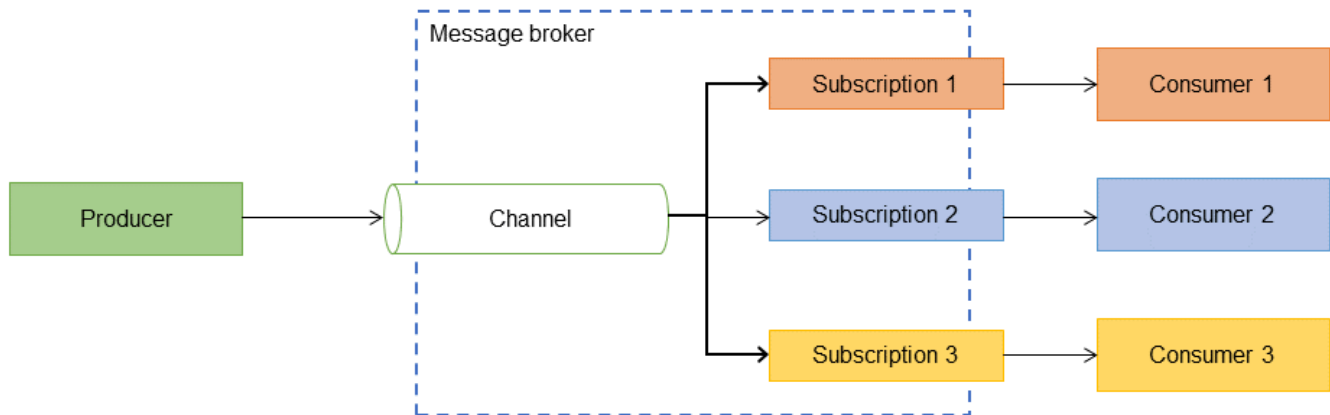
- Reprodução de mensagens: os recursos de reprodução de mensagens podem depender da infraestrutura de mensagens. Você também pode fornecer implementações personalizadas, dependendo do caso de uso.
- Filas de mensagens não entregues: em um sistema postal, um escritório de mensagens não entregues é uma instalação para processar correspondências não entregues. Em [mensagens pub/ass](#), uma fila de mensagens não entregues (DLQ) é uma fila para mensagens que não podem ser entregues a um endpoint assinante.

Implementação

Arquitetura de alto nível

Em um padrão publicar/assinar, o subsistema de mensagens assíncronas conhecido como agente de mensagens ou roteador acompanha as assinaturas. Quando um produtor publica um evento, a infraestrutura de mensagens envia uma mensagem para cada consumidor. Depois que uma mensagem é enviada aos assinantes, ela é removida da infraestrutura de mensagens para que não possa ser reproduzida e os novos assinantes não vejam o evento. Os agentes de mensagens ou roteadores separam o produtor de eventos dos consumidores de mensagens da seguinte forma:

- Fornecer um canal de entrada para o produtor publicar eventos que são empacotados em mensagens, usando um formato de mensagem definido.
- Criação de um canal de saída individual por assinatura. Uma assinatura é a conexão do consumidor, na qual ele recebe mensagens de eventos associadas a um canal de entrada específico.
- Copiar mensagens do canal de entrada para o canal de saída para todos os consumidores quando o evento for publicado.



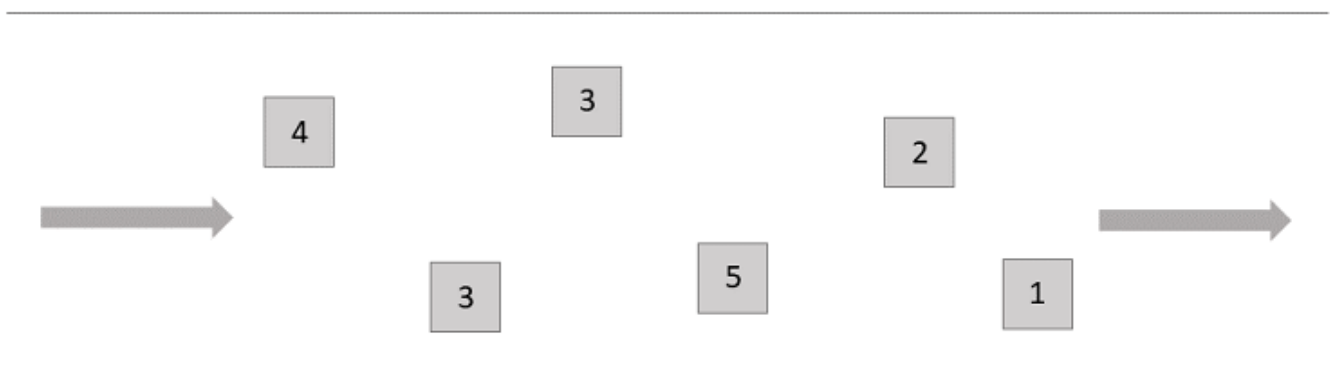
Implementação usando serviços da AWS

Amazon SNS

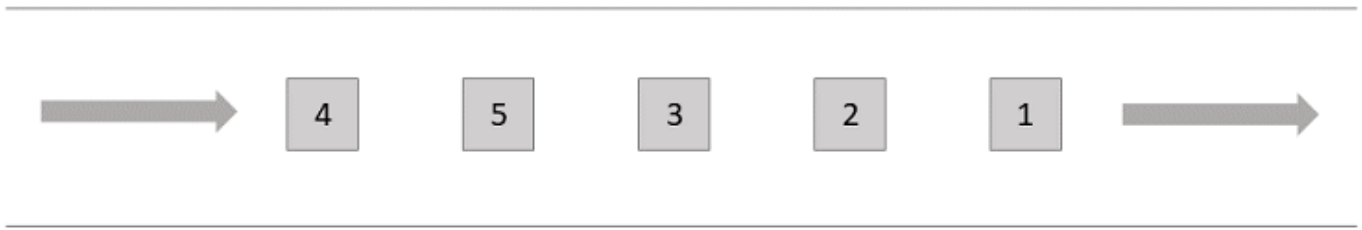
O Amazon SNS é um serviço de publicador-assinante totalmente gerenciado que fornece mensagens de aplicação para aplicação (A2A) para dissociar aplicações distribuídas. Ele também fornece mensagens de aplicação para pessoa (A2P) para enviar SMS, e-mail e outras notificações push.

O Amazon SNS fornece dois tipos de tópicos: padrão e primeiro a entrar, primeiro a sair (FIFO).

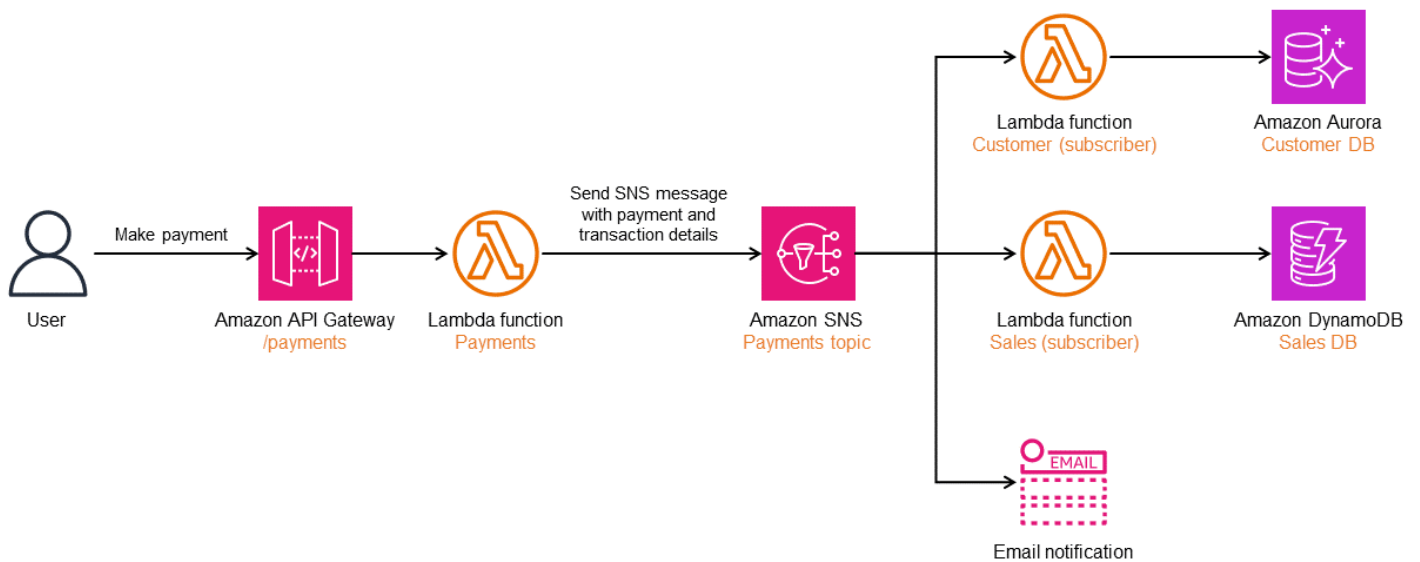
- Os tópicos padrão oferecem suporte a um número ilimitado de mensagens por segundo e oferecem o melhor esforço para ordenação e deduplicação.



- Os tópicos do FIFO fornecem ordenação e deduplicação rigorosas e oferecem suporte a até 300 mensagens por segundo ou 10 MB por segundo por tópico do FIFO (o que ocorrer primeiro).



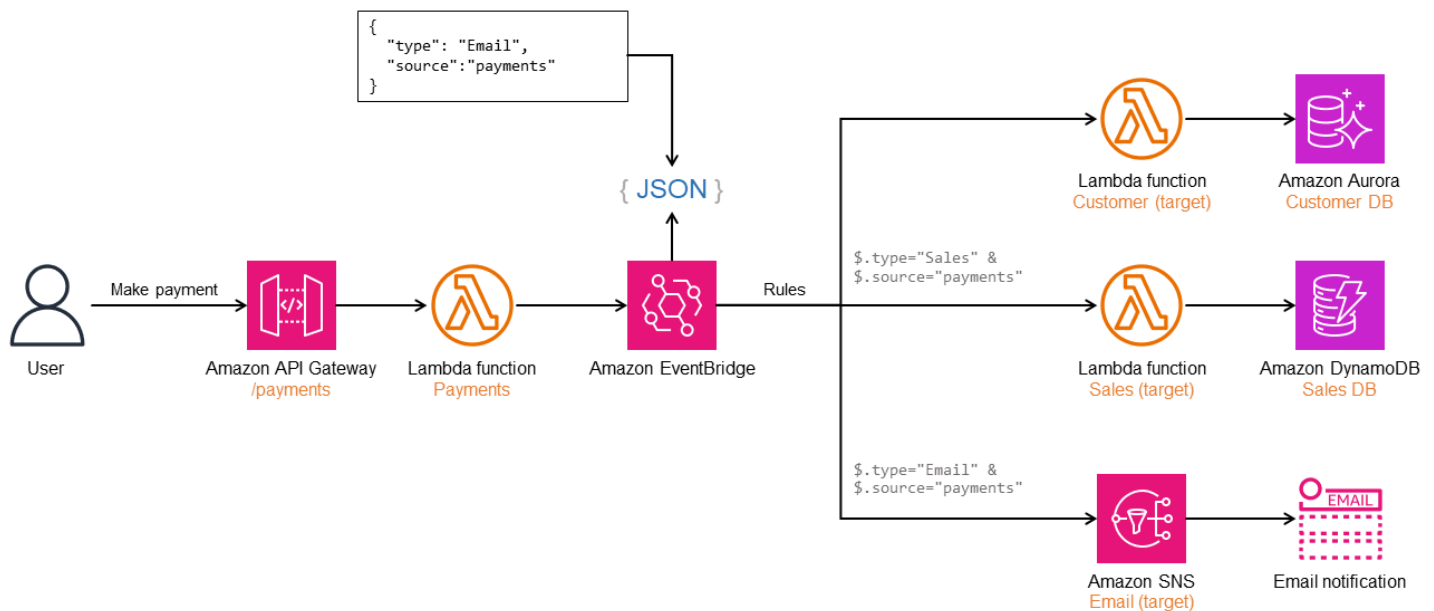
A ilustração a seguir mostra como você pode usar o Amazon SNS para implementar o padrão publicar/assinar. Depois que um usuário faz um pagamento, uma mensagem do SNS é enviada pela função Payments do Lambda para o tópico Payments do SNS. Este tópico do SNS tem três assinantes. Cada assinante recebe uma cópia da mensagem e a processa.



Amazon EventBridge

Você pode usar o Amazon EventBridge quando precisar de um roteamento mais complexo de mensagens de vários produtores em diferentes protocolos para consumidores assinantes ou de assinaturas diretas e dispersas (fanout). O EventBridge também oferece suporte a roteamento, filtragem, sequenciamento e divisão ou agregação com base em conteúdo. Na ilustração a seguir, o EventBridge é usado para criar uma versão do padrão publicar/assinar na qual os assinantes são definidos usando regras de eventos. Depois que um usuário faz um pagamento, a função Payments do Lambda envia uma mensagem para o EventBridge usando o barramento de eventos padrão com

base em um esquema personalizado que tem três regras apontando para destinos diferentes. Cada microsserviço processa as mensagens e executa as ações necessárias.



Workshop

- [Criação de arquiteturas orientadas por eventos na AWS](#)
- [Enviar Notificações Fanout de Eventos por meio do Amazon Simple Queue Service \(Amazon SQS\) e do Amazon Simple Notification Service \(Amazon SNS\)](#)

Referências do blog

- [Escolher entre serviços de mensagens para aplicativos sem servidor](#)
- [Projetar aplicativos duráveis sem servidor com DLQs para Amazon SNS, Amazon SQS e AWS Lambda](#)
- [Simplificar suas mensagens pub/ass com a filtragem de mensagens do Amazon SNS](#)

Conteúdo relacionado

- [Atributos de mensagens pub/ass](#)

Tente novamente com o padrão de recuo

Intenção

O padrão de repetição com recuo melhora a estabilidade da aplicação ao repetir de forma transparente as operações que falham devido a erros transitórios.

Motivação

Em arquiteturas distribuídas, erros transitórios podem ser causados por controle de utilização do serviço, perda temporária de conectividade de rede ou indisponibilidade temporária do serviço. Repetir automaticamente as operações que falham devido a esses erros transitórios melhora a experiência do usuário e a resiliência da aplicação. No entanto, novas tentativas frequentes podem sobrecarregar a largura de banda da rede e causar contenção. O recuo exponencial é uma técnica em que as operações são repetidas aumentando os tempos de espera para um número específico de tentativas de repetição.

Aplicabilidade

Use o padrão de repetição com recuo quando:

- Seus serviços frequentemente controlam a utilização de solicitações para evitar sobrecarga, resultando em uma exceção 429 Muitas solicitações no processo de chamada.
- A rede é um participante invisível em arquiteturas distribuídas, e problemas temporários de rede resultam em falhas.
- O serviço que está sendo chamado está temporariamente indisponível, causando falhas. Tentativas frequentes podem causar degradação do serviço, a menos que você introduza um tempo limite de recuo usando esse padrão.

Problemas e considerações

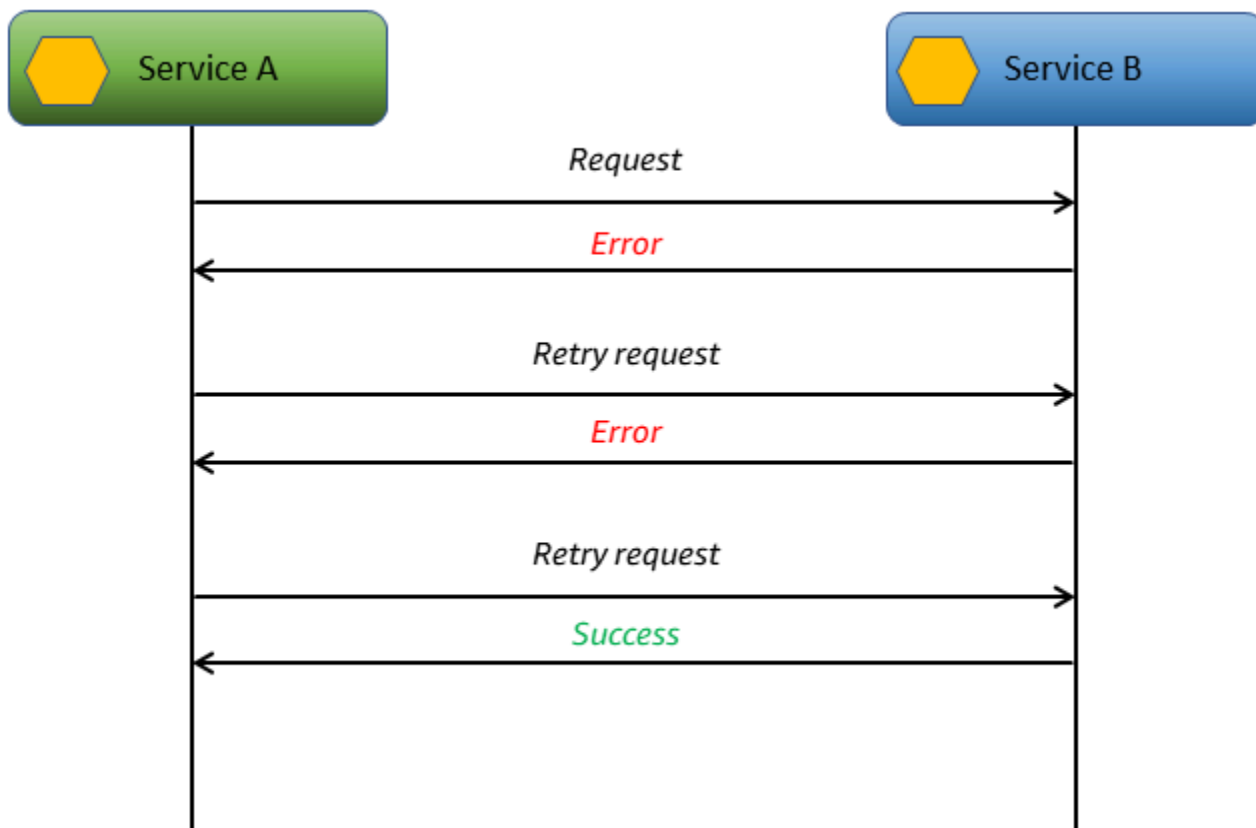
- Idempotência: se várias chamadas para o método tiverem o mesmo efeito de uma única chamada no estado do sistema, a operação será considerada idempotente. As operações devem ser idempotentes quando você usa o padrão de repetição com recuo. Caso contrário, atualizações parciais podem corromper o estado do sistema.

- Largura de banda da rede: a degradação do serviço poderá ocorrer se muitas tentativas ocuparem a largura de banda da rede, levando a tempos de resposta lentos.
- Cenários de falha rápida: para erros não transitórios, se você puder determinar a causa da falha, é mais eficiente antecipar-se à falha usando o padrão do disjuntor.
- Taxa de recuo: a introdução do recuo exponencial pode ter um impacto no tempo limite do serviço, resultando em maiores tempos de espera para o usuário final.

Implementação

Arquitetura de alto nível

O diagrama a seguir ilustra como o Serviço A pode repetir as chamadas para o Serviço B até que uma resposta bem-sucedida seja retornada. Se o Serviço B não retornar uma resposta bem-sucedida após algumas tentativas, o Serviço A poderá parar de tentar novamente e retornar uma falha ao chamador.

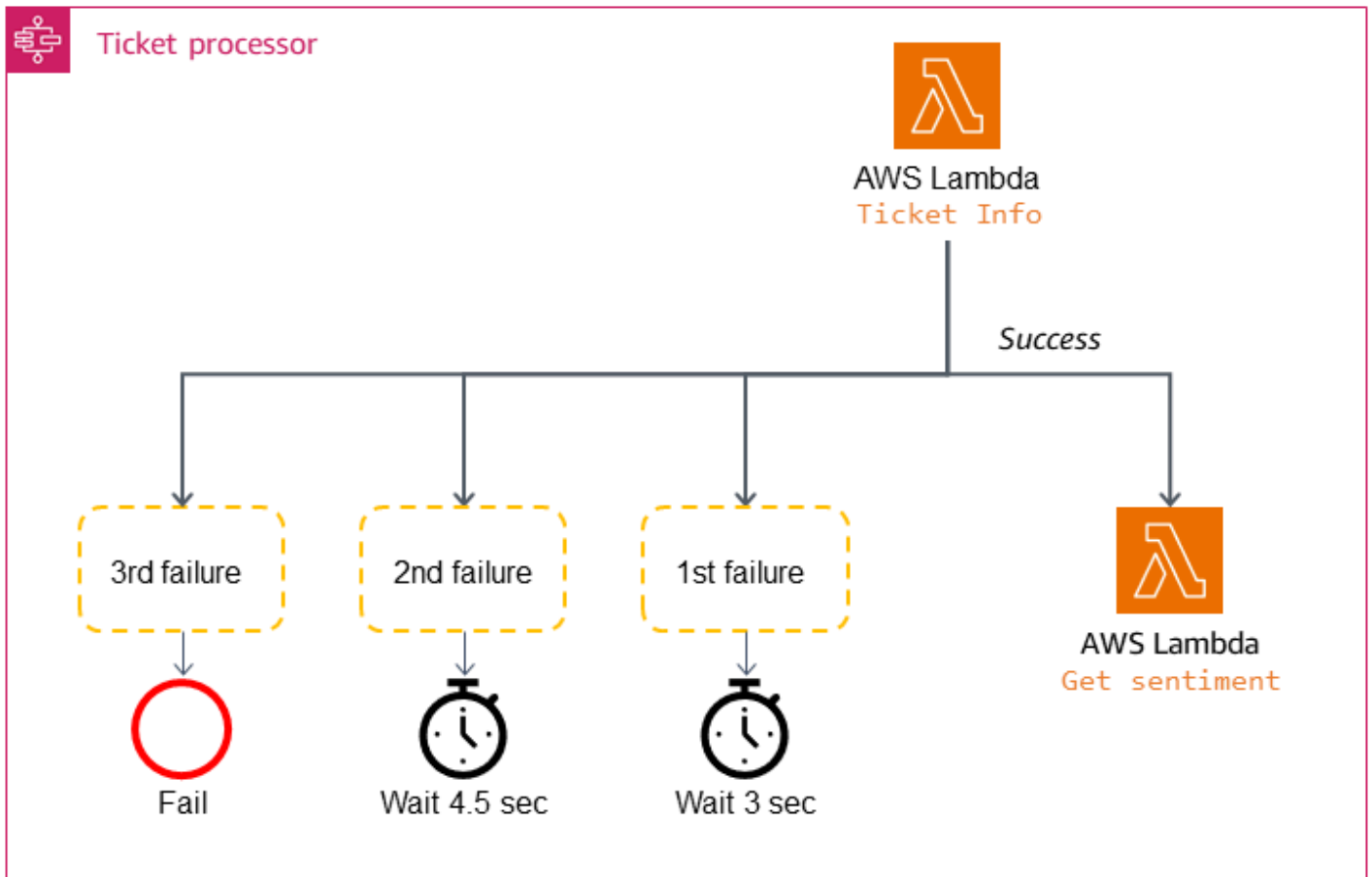


Implementação usando serviços AWS

O diagrama a seguir mostra um fluxo de trabalho de processamento de tíquetes em uma plataforma de suporte ao cliente. Os tíquetes de clientes insatisfeitos são acelerados escalando automaticamente a prioridade dos tíquetes. A função `Ticket info` do Lambda extrai os detalhes do tíquete e chama a função `Get sentiment` do Lambda. A função `Get sentiment` do Lambda verifica os sentimentos do cliente passando a descrição para o [Amazon Comprehend](#) (não mostrado).

Se a chamada para a função `Get sentiment` do Lambda falhar, o fluxo de trabalho repetirá a operação três vezes. O AWS Step Functions permite o recuo exponencial, permitindo que você configure o valor do recuo.

Neste exemplo, no máximo três novas tentativas são configuradas com um multiplicador de aumento de 1,5 segundo. Se a primeira tentativa ocorrer após 3 segundos, a segunda tentativa ocorrerá após $3 \times 1,5$ segundo = 4,5 segundos, e a terceira tentativa ocorrerá após $4,5 \times 1,5$ segundo = 6,75 segundos. Se a terceira tentativa não for bem-sucedida, o fluxo de trabalho falhará. A lógica de recuo não requer nenhum código personalizado, ela é fornecida como uma configuração pelo AWS Step Functions.



Código de exemplo

O código a seguir mostra a implementação do padrão de repetição com recuo.

```
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);
}
```

Repositório GitHub

Para uma implementação completa da arquitetura de amostra desse padrão, consulte o repositório do GitHub em <https://github.com/aws-samples/retry-with-backoff>.

Conteúdo relacionado

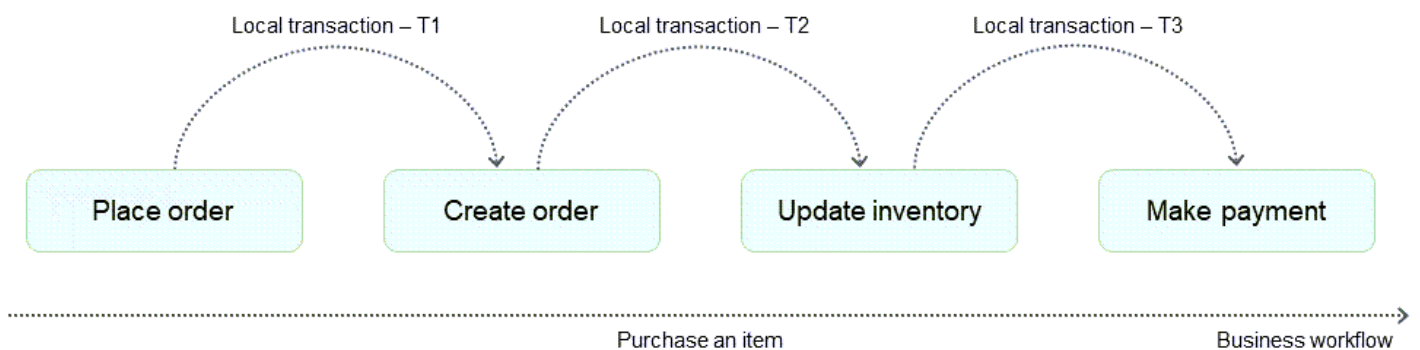
- [Timeouts, retries, and backoff with jitter](#) (Amazon Builders' Library)

Padrões saga

Uma saga consiste em uma sequência de transações locais. Cada transação local em uma saga atualiza o banco de dados e aciona a próxima transação local. Se uma transação falhar, a saga executa transações compensatórias para reverter as alterações do banco de dados feitas pelas transações anteriores.

Essa sequência de transações locais ajuda a alcançar um fluxo de trabalho comercial usando princípios de continuação e compensação. O princípio de continuação decide a recuperação futura do fluxo de trabalho, enquanto o princípio de compensação decide a recuperação retroativa. Se a atualização falhar em qualquer etapa da transação, a saga publica um evento para continuação (para repetir a transação) ou compensação (para voltar ao estado anterior dos dados). Isso garante que a integridade dos dados seja mantida e consistente em todos os repositórios de dados.

Por exemplo, quando um usuário compra um livro de um varejista on-line, o processo consiste em uma sequência de transações, como criação do pedido, atualização do estoque, pagamento e envio, que representa um fluxo de trabalho comercial. Para concluir esse fluxo de trabalho, a arquitetura distribuída emite uma sequência de transações locais para criar um pedido no banco de dados de pedidos, atualizar o banco de dados de estoque e atualizar o banco de dados de pagamento. Quando o processo é bem-sucedido, essas transações são invocadas sequencialmente para concluir o fluxo de trabalho comercial, conforme mostra o diagrama a seguir. No entanto, se alguma dessas transações locais falhar, o sistema deverá ser capaz de decidir sobre a próxima etapa apropriada, ou seja, uma recuperação para a frente ou uma recuperação para trás.



Os dois cenários a seguir ajudam a determinar se a próxima etapa é a recuperação para a frente ou para trás:

- Falha no nível da plataforma, em que algo dá errado com a infraestrutura subjacente e faz com que a transação falhe. Nesse caso, o padrão saga pode realizar uma recuperação para a frente tentando novamente a transação local e continuando o processo comercial.
- Falha no nível do aplicativo, em que o serviço de pagamento falha devido a um pagamento inválido. Nesse caso, o padrão saga pode realizar uma recuperação para trás emitindo uma transação compensatória para atualizar o estoque e os bancos de dados de pedidos e restabelecer seu estado anterior.

O padrão saga gerencia o fluxo de trabalho comercial e garante que um estado final desejável seja alcançado por meio da recuperação para a frente. Em caso de falhas, ele reverte as transações locais usando a recuperação reversa para evitar problemas de consistência de dados.

O padrão saga tem duas variantes: coreografia e orquestração.

Coreografia saga

O padrão da coreografia saga depende dos eventos publicados pelos microsserviços. Os participantes da saga (microsserviços) se inscrevem nos eventos e agem com base nos gatilhos do evento. Por exemplo, o serviço de pedidos no diagrama a seguir emite um evento `OrderPlaced`. O serviço de estoque se inscreve nesse evento e atualiza o estoque quando o evento `OrderPlaced` é emitido. Da mesma forma, os serviços participantes atuam com base no contexto do evento emitido.

O padrão coreográfico da saga é adequado quando há apenas alguns participantes na saga e você precisa de uma implementação simples sem um único ponto de falha. Quando mais participantes são adicionados, mais difícil fica rastrear as dependências entre os participantes usando esse padrão.



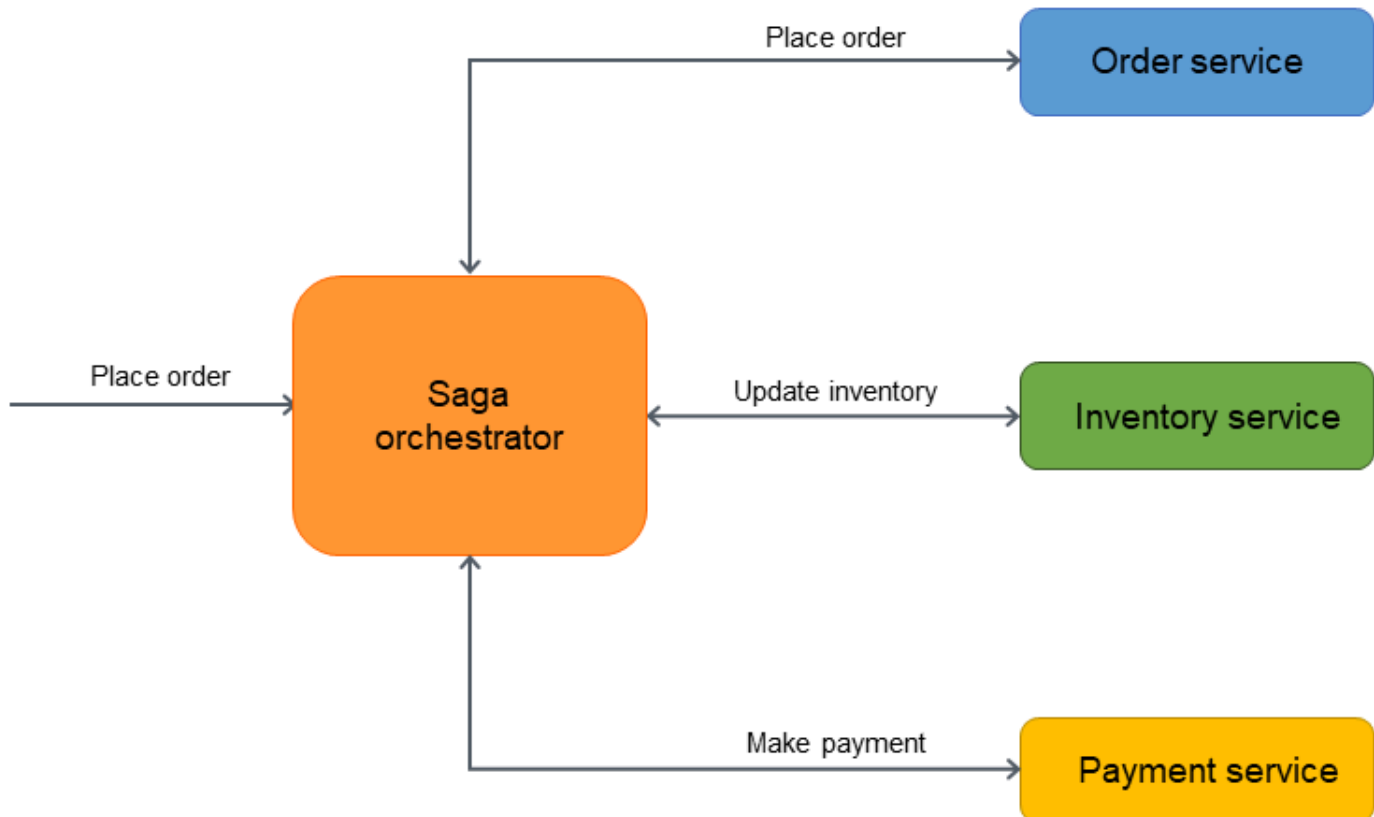
Para uma análise detalhada, consulte a seção de [coreografia de Saga](#) deste guia.

Orquestração da saga

O padrão de orquestração saga tem um coordenador central chamado orquestrador. O orquestrador da saga gerencia e coordena todo o ciclo de vida da transação. Ele está ciente da série de etapas a serem executadas para concluir a transação. Para executar uma etapa, ele envia uma mensagem ao microsserviço participante para realizar a operação. O microsserviço participante conclui a

operação e envia uma mensagem de volta ao orquestrador. Com base na mensagem que recebe, o orquestrador decide qual microserviço executar em seguida na transação.

O padrão de orquestração saga é adequado quando há muitos participantes, e é necessário um acoplamento fraco entre os participantes da saga. O orquestrador encapsula a complexidade da lógica ao tornar os participantes fracamente acoplados. No entanto, o orquestrador pode se tornar um único ponto de falha porque controla todo o fluxo de trabalho.



Para uma análise detalhada, consulte a seção de [orquestração saga](#) deste guia.

Padrão da coreografia saga

Intenção

O padrão de coreografia saga ajuda a preservar a integridade dos dados em transações distribuídas que abrangem vários serviços usando assinaturas de eventos. Em uma transação distribuída, vários serviços podem ser chamados antes que uma transação seja concluída. Quando os serviços armazenam dados em diferentes repositórios de dados, pode ser difícil manter a consistência de dados neles.

Motivação

Uma transação é uma única unidade de trabalho que pode envolver várias etapas, em que todas as etapas são completamente executadas ou nenhuma etapa é executada, resultando em um armazenamento de dados que mantém seu estado consistente. Os termos atomicidade, consistência, isolamento e durabilidade (ACID) definem as propriedades de uma transação. Os bancos de dados relacionais fornecem transações ACID para manter a consistência de dados.

Para manter a consistência em uma transação, os bancos de dados relacionais usam o método de confirmação em duas fases (2PC). Isso consiste em uma fase de preparação e uma fase de confirmação.

- Na fase de preparação, o processo de coordenação solicita que os processos participantes da transação (participantes) prometam confirmar ou reverter a transação.
- Na fase de confirmação, o processo de coordenação solicita que os participantes confirmem a transação. Se os participantes não concordarem em se comprometer na fase de preparação, a transação será revertida.

Em sistemas distribuídos que seguem um padrão database-per-service de design, a confirmação em duas fases não é uma opção. Isso ocorre porque cada transação é distribuída em vários bancos de dados e não há um único controlador que possa coordenar um processo semelhante à confirmação em duas fases em repositórios de dados relacionais. Nesse caso, uma solução é usar o padrão de coreografia saga.

Aplicabilidade

Use o padrão de coreografia saga quando:

- Seu sistema exige integridade e consistência de dados em transações distribuídas que abrangem vários repositórios de dados.
- O armazenamento de dados (por ex., um banco de dados NoSQL) não fornece 2PC para fornecer transações ACID; você precisa atualizar várias tabelas em uma única transação, e implementar 2PC dentro dos limites do aplicativo seria uma tarefa complexa.
- Um processo de controle central que gerencia as transações dos participantes pode se tornar um único ponto de falha.
- Os participantes da saga são serviços independentes e precisam ser vagamente acoplados.
- Há comunicação entre contextos limitados em um domínio comercial.

Problemas e considerações

- **Complexidade:** à medida que o número de microsserviços aumenta, a coreografia saga pode tornar-se difícil de gerenciar devido ao número de interações entre os microsserviços. Além disso, transações compensatórias e novas tentativas adicionam complexidades ao código do aplicativo, o que pode resultar em sobrecarga de manutenção. A coreografia estará adequada quando houver poucos participantes na saga e você precisar de uma implementação simples sem um único ponto de falha. Quando mais participantes são adicionados, mais difícil fica rastrear as dependências entre os participantes usando esse padrão.
- **Implementação resiliente:** na coreografia Saga, é mais difícil implementar tempos limite, novas tentativas e outros padrões de resiliência globalmente, em comparação com a orquestração Saga. A coreografia deve ser implementada em componentes individuais em vez de em nível de orquestrador.
- **Dependências cíclicas:** os participantes consomem mensagens publicadas uns pelos outros. Isso pode resultar em dependências cíclicas, levando a complexidades de código e sobrecargas de manutenção, além de possíveis impasses.
- **Problema de gravação dupla:** o microsserviço precisa atualizar atômicamente o banco de dados e publicar um evento. A falha de qualquer operação pode levar a um estado inconsistente. Uma maneira de resolver isso é usar o padrão de caixa de [saída transacional](#).
- **Preservando eventos:** os participantes da saga agem com base nos eventos publicados. É importante salvar os eventos na ordem em que eles ocorrem para fins de auditoria, depuração e repetição. Você pode usar o [padrão de fornecimento de eventos](#) para persistir os eventos em um repositório de eventos, caso seja necessária uma repetição do estado do sistema para restaurar a consistência de dados. Os repositórios de eventos também podem ser usados para fins de auditoria e solução de problemas, pois refletem todas as alterações no sistema.
- **Consistência eventual:** o processamento sequencial de transações locais resulta em consistência eventual, o que pode ser um desafio em sistemas que exigem consistência forte. Você pode resolver esse problema definindo as expectativas de suas equipes comerciais em relação ao modelo de consistência ou reavaliando o caso de uso e migrando para um banco de dados que forneça consistência forte.
- **Idempotência:** os participantes da saga precisam ser idempotentes para permitir a execução repetida em caso de falhas transitórias causadas por falhas inesperadas e falhas do orquestrador.
- **Isolamento de transação:** o padrão saga não tem isolamento de transação, que é uma das quatro propriedades nas transações ACID. O [grau de isolamento](#) de uma transação determina quantas ou quão poucas outras transações simultâneas podem afetar os dados nos quais ela opera. A

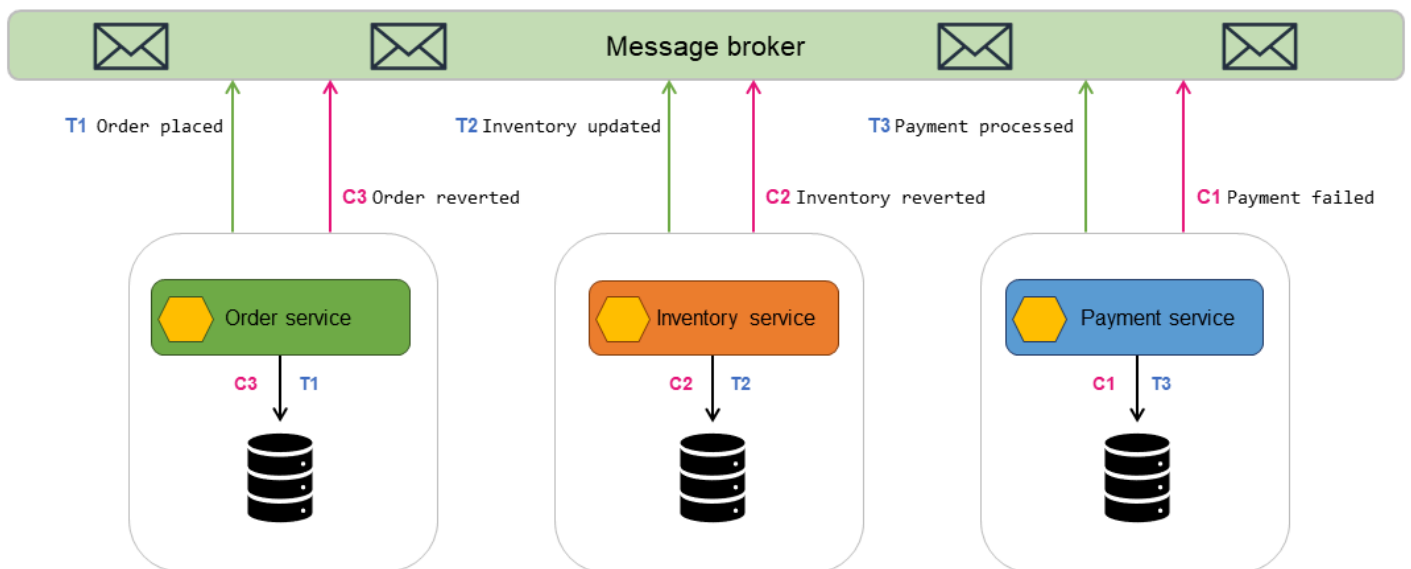
orquestração simultânea de transações pode levar a dados obsoletos. Recomendamos usar o bloqueio semântico para lidar com esses cenários.

- **Observabilidade:** observabilidade se refere ao registro e rastreamento detalhados para solucionar problemas no processo de implementação e orquestração. Isso se torna importante quando o número de participantes da saga aumenta, resultando em complexidades na depuração. End-to-end monitoramento e reportagem são mais difíceis de alcançar na coreografia de saga, em comparação com a orquestração de saga.
- **Problemas de latência:** transações compensatórias podem adicionar latência ao tempo geral de resposta quando a saga consiste em várias etapas. Se as transações fizerem chamadas síncronas, isso pode aumentar ainda mais a latência.

Implementação

Arquitetura de alto nível

No diagrama de arquitetura a seguir, a coreografia da saga tem três participantes: o serviço de pedidos, o serviço de estoque e o serviço de pagamento. São necessárias três etapas para concluir a transação: T1, T2 e T3. Três transações compensatórias restauram os dados ao estado inicial: C1, C2 e C3.



- O serviço de pedidos executa uma transação local, T1, que atualiza atômica o banco de dados e publica uma mensagem `Order placed` no agente de mensagens.

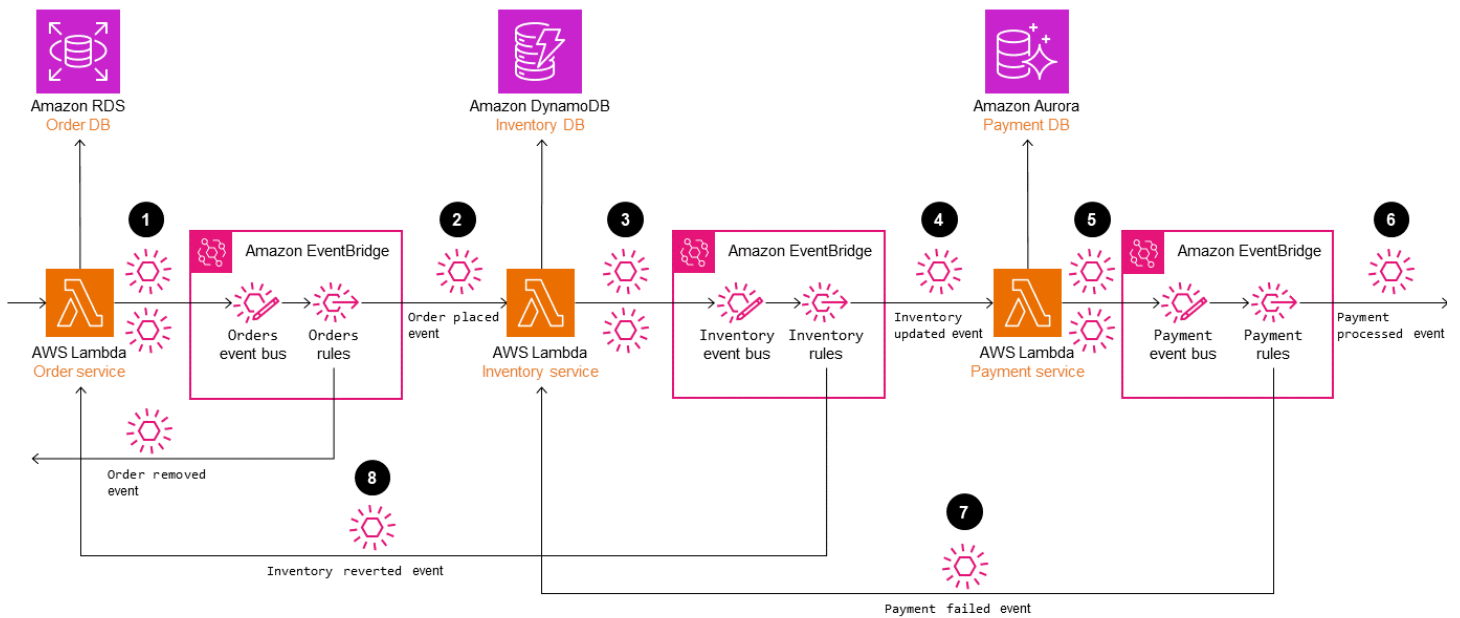
- O serviço de estoque assina as mensagens do serviço de pedidos e recebe a mensagem de que um pedido foi criado.
- O serviço de estoque executa uma transação local, T2, que atualiza atômicamente o banco de dados e publica uma mensagem `Inventory updated` no agente de mensagens.
- O serviço de pagamento assina as mensagens do serviço de estoque e recebe a mensagem de que o estoque foi atualizado.
- O serviço de pagamento executa uma transação local, T3, que atualiza atômicamente o banco de dados com detalhes de pagamento e publica uma mensagem `Payment processed` para o agente de mensagens.
- Se o pagamento falhar, o serviço de pagamento executa uma transação compensatória, C1, que reverte atômicamente o pagamento no banco de dados e publica uma mensagem `Payment failed` para o agente de mensagens.
- As transações compensatórias C2 e C3 são executadas para restaurar a consistência dos dados.

Implementação usando serviços da AWS

Você pode implementar o padrão de coreografia da saga usando a Amazon. EventBridge EventBridge usa eventos para conectar os componentes do aplicativo. Ele processa eventos por meio de tubos ou barramentos de eventos. Um barramento de eventos é um roteador que recebe [eventos](#) e os entrega a zero ou mais destinos, ou alvos.. [As regras](#) associadas ao barramento de eventos avaliam os eventos à medida que eles chegam e os enviam aos [alvos](#) para processamento.

Na seguinte arquitetura:

- Os microsserviços — serviço de pedidos, serviço de estoque e serviço de pagamento — são implementados como funções do Lambda.
- Existem três EventBridge ônibus personalizados: ônibus para `Orders` eventos, ônibus para `Inventory` eventos e ônibus para `Payment` eventos.
- As regras `Orders`, regras `Inventory` e regras `Payment` correspondem aos eventos que são enviados para o barramento de eventos correspondente e invocam as funções do Lambda.



Em um cenário bem-sucedido, quando um pedido é feito:

1. O serviço de pedidos processa a solicitação e envia o evento para o barramento de eventos Orders.
2. As regras Orders correspondem aos eventos e iniciam o serviço de estoque.
3. O serviço de estoque atualiza o estoque e envia o evento para o barramento de eventos Inventory.
4. As regras Inventory correspondem aos eventos e iniciam o serviço de pagamento.
5. O serviço de pagamento processa o pagamento e envia o evento para o barramento de eventos Payment.
6. As regras Payment correspondem aos eventos e enviam a notificação do evento Payment processed ao receptor.

Como alternativa, quando há um problema no processamento do pedido, EventBridge as regras iniciam as transações compensatórias para reverter as atualizações de dados para manter a consistência e a integridade dos dados.

7. Se o pagamento falhar, as regras Payment processam o evento e iniciam o serviço de estoque. O serviço de inventário executa transações compensatórias para reverter o estoque.

- Quando o estoque é revertido, o serviço de estoque envia o evento `Inventory reverted` para o barramento de eventos `Inventory`. Esse evento é processado por regras `Inventory`. Ele inicia o serviço de pedidos, que executa a transação compensatória para remover o pedido.

Conteúdo relacionado

- [Padrão de orquestração da saga](#)
- [Padrão de caixa de saída transacional](#)
- [Tente novamente com o padrão de recuo](#)

Padrão de orquestração saga

Intenção

O padrão de orquestração saga usa um coordenador central (orquestrador) para ajudar a preservar a integridade dos dados em transações distribuídas que abrangem vários serviços. Em uma transação distribuída, vários serviços podem ser chamados antes que uma transação seja concluída. Quando os serviços armazenam dados em diferentes repositórios de dados, pode ser difícil manter a consistência de dados neles.

Motivação

Uma transação é uma única unidade de trabalho que pode envolver várias etapas, em que todas as etapas são completamente executadas ou nenhuma etapa é executada, resultando em um armazenamento de dados que mantém seu estado consistente. Os termos atomicidade, consistência, isolamento e durabilidade (ACID) definem as propriedades de uma transação. Os bancos de dados relacionais fornecem transações ACID para manter a consistência de dados.

Para manter a consistência em uma transação, os bancos de dados relacionais usam o método de confirmação em duas fases (2PC). Isso consiste em uma fase de preparação e uma fase de confirmação.

- Na fase de preparação, o processo de coordenação solicita que os processos participantes da transação (participantes) prometam confirmar ou reverter a transação.
- Na fase de confirmação, o processo de coordenação solicita que os participantes confirmem a transação. Se os participantes não concordarem em se comprometer na fase de preparação, a transação será revertida.

Em sistemas distribuídos que seguem um padrão database-per-service de design, a confirmação em duas fases não é uma opção. Isso ocorre porque cada transação é distribuída em vários bancos de dados e não há um único controlador que possa coordenar um processo semelhante à confirmação em duas fases em repositórios de dados relacionais. Nesse caso, uma solução é usar o padrão de orquestração saga.

Aplicabilidade

Use o padrão de orquestração saga quando:

- Seu sistema exige integridade e consistência de dados em transações distribuídas que abrangem vários repositórios de dados.
- O armazenamento de dados não fornece 2PC para fornecer transações ACID, e implementar 2PC dentro dos limites do aplicativo é uma tarefa complexa.
- Você tem bancos de dados NoSQL, que não fornecem transações ACID, e precisa atualizar várias tabelas em uma única transação.

Problemas e considerações

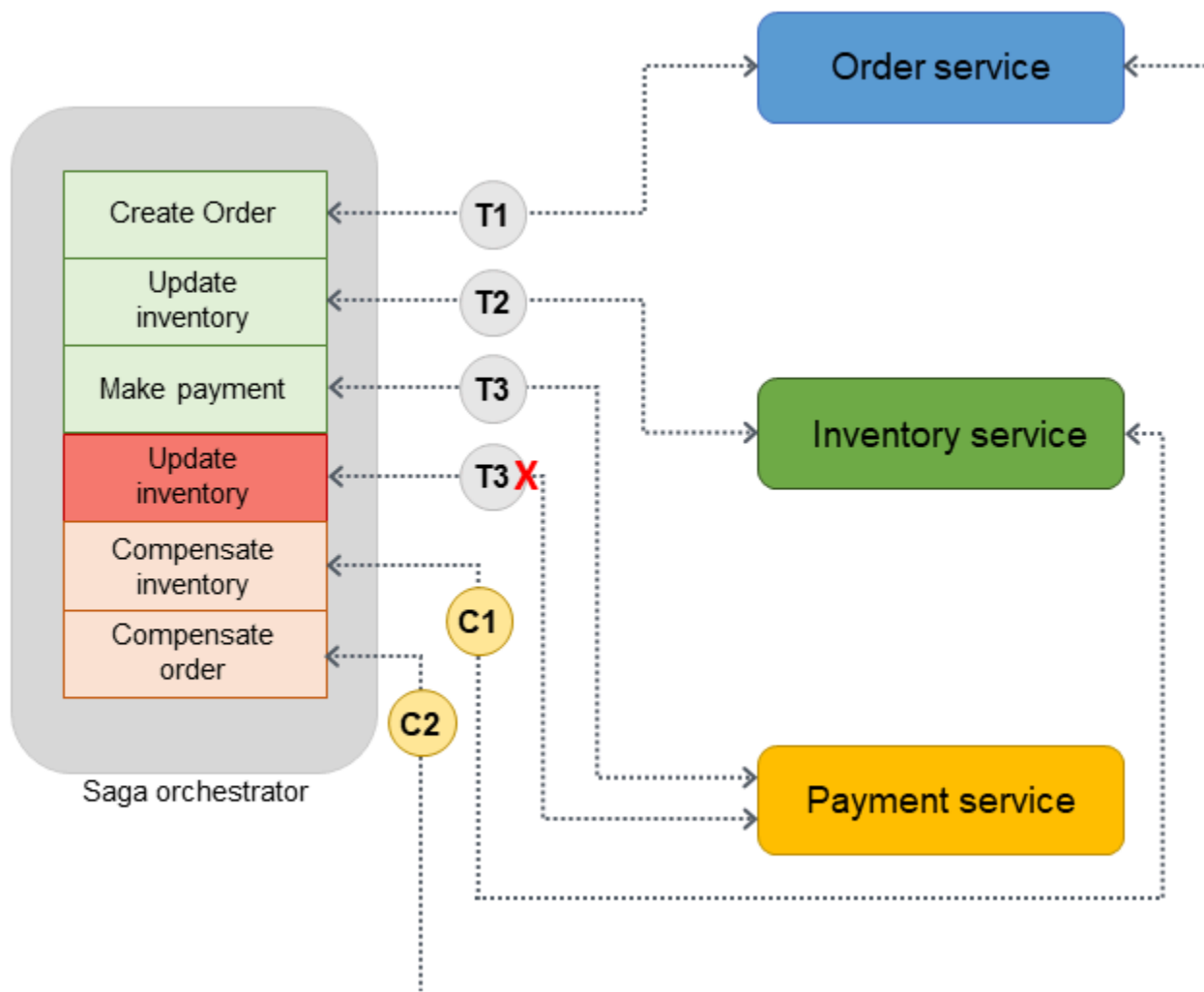
- Complexidade: transações compensatórias e novas tentativas adicionam complexidades ao código do aplicativo, o que pode resultar em sobrecarga de manutenção.
- Consistência eventual: o processamento sequencial de transações locais resulta em consistência eventual, o que pode ser um desafio em sistemas que exigem consistência forte. Você pode resolver esse problema definindo as expectativas de suas equipes comerciais em relação ao modelo de consistência ou mudando para um armazenamento de dados que forneça uma consistência forte.
- Idempotência: os participantes da saga precisam ser idempotentes para permitir a execução repetida em caso de falhas transitórias causadas por falhas inesperadas e falhas do orquestrador.
- Isolamento de transações: saga não tem isolamento de transações. A orquestração simultânea de transações pode levar a dados obsoletos. Recomendamos usar o bloqueio semântico para lidar com esses cenários.
- Observabilidade: observabilidade se refere ao registro e ao rastreamento detalhados para solucionar problemas no processo de execução e orquestração. Isso torna-se importante quando o número de participantes da saga aumenta, resultando em complexidades na depuração.
- Problemas de latência: transações compensatórias podem adicionar latência ao tempo geral de resposta quando a saga consiste em várias etapas. Evite chamadas síncronas nesses casos.

- Ponto único de falha: o orquestrador pode se tornar um ponto único de falha porque coordena toda a transação. Em alguns casos, o padrão de coreografia da saga é preferido por causa dessa questão.

Implementação

Arquitetura de alto nível

No diagrama de arquitetura a seguir, o orquestrador saga tem três participantes: o serviço de pedidos, o serviço de estoque e o serviço de pagamento. São necessárias três etapas para concluir a transação: T1, T2 e T3. O orquestrador saga está ciente das etapas e as executa na ordem necessária. Quando a etapa T3 falha (falha no pagamento), o orquestrador executa as transações compensatórias C1 e C2 para restaurar os dados ao estado inicial.



Você pode usar [AWS Step Functions](#) para implementar a orquestração saga quando a transação é distribuída em vários bancos de dados.

Implementação usando AWS serviços

A solução de amostra usa o fluxo de trabalho padrão em Step Functions para implementar o padrão de orquestração saga.



Quando um cliente chama a API, a função do Lambda é invocada e o pré-processamento ocorre na função do Lambda. A função inicia o fluxo de trabalho do Step Functions para começar a processar a transação distribuída. Se o pré-processamento não for necessário, você poderá [iniciar o fluxo de trabalho do Step Functions diretamente](#) do API Gateway sem usar a função do Lambda.

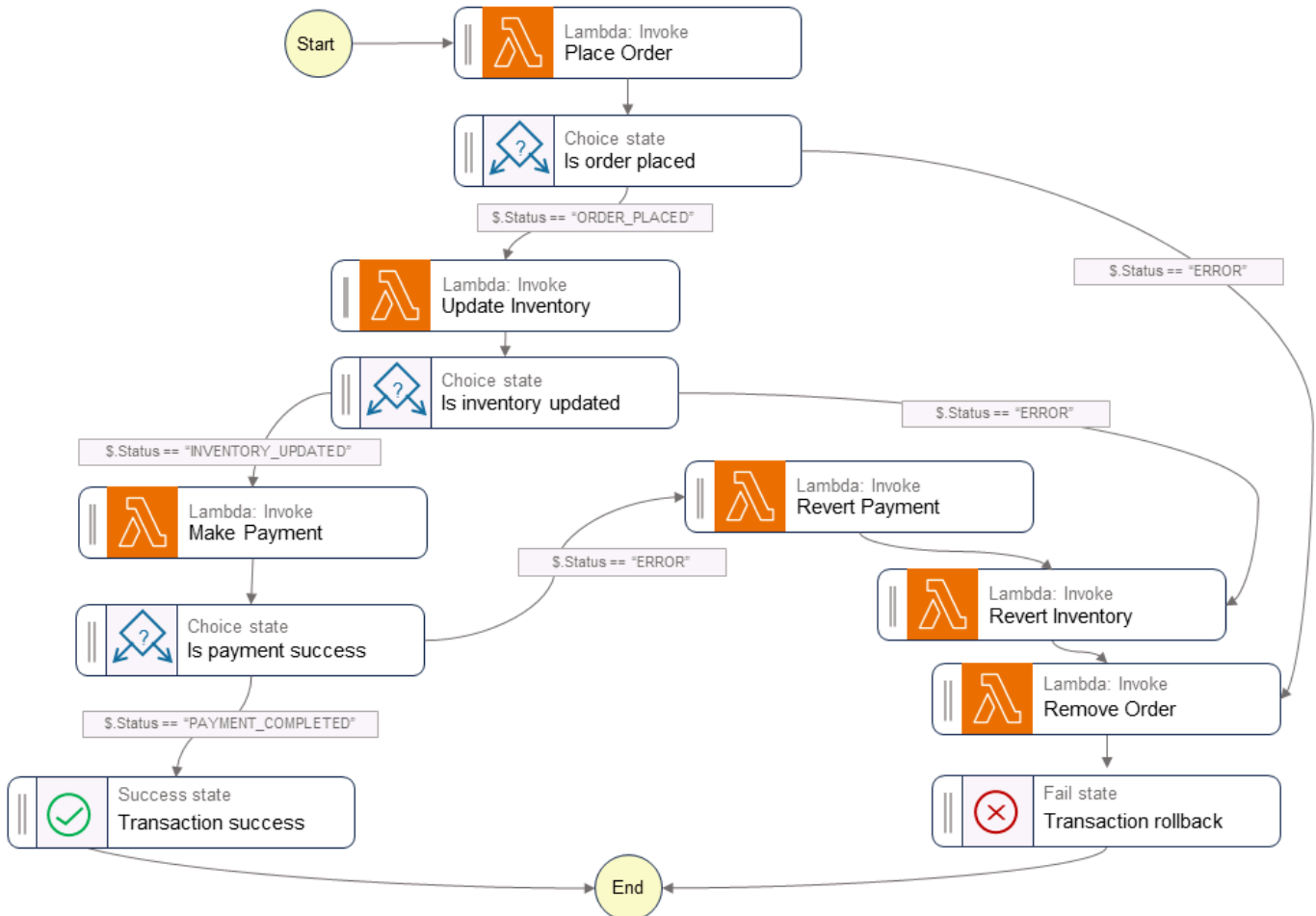
O uso do Step Functions atenua o problema do ponto único de falha, que é inerente à implementação do padrão de orquestração saga. O Step Functions tem a tolerância a falhas integrada e mantém a capacidade de serviço em várias zonas de disponibilidade em cada região da AWS para proteger as aplicações contra falhas individuais em máquinas ou data centers. Isso ajuda a garantir a alta disponibilidade do serviço em si e do fluxo de trabalho do aplicativo que ele opera.

O fluxo de trabalho do Step Functions

A máquina de estado Step Functions permite que você configure os requisitos de fluxo de controle com base em decisão para a implementação do padrão. O fluxo de trabalho do Step Functions chama os serviços individuais para colocação de pedidos, atualização de estoque e processamento de pagamentos para concluir a transação e envia uma notificação de evento para processamento adicional. O fluxo de trabalho do Step Functions atua como orquestrador para coordenar as transações. Se o fluxo de trabalho contiver algum erro, o orquestrador executará as transações compensatórias para garantir que a integridade dos dados seja mantida em todos os serviços.

O diagrama a seguir mostra as etapas que são executadas no fluxo de trabalho do Step Functions. As etapas `Place Order`, `Update Inventory`, e `Make Payment` indicam o caminho do sucesso. O pedido é feito, o estoque é atualizado e o pagamento é processado antes que um `Success` estado seja devolvido ao chamador.

As funções `Revert Payment`, `Revert Inventory`, e `Remove Order` do Lambda indicam as transações compensatórias que o orquestrador executa quando alguma etapa do fluxo de trabalho falha. Se o fluxo de trabalho falhar na etapa `Update Inventory`, o orquestrador chama as etapas `Revert Inventory` e `Remove Order` e antes de retornar um estado `Fail` ao chamador. Essas transações compensatórias garantem que a integridade dos dados seja mantida. O estoque volta ao nível original e o pedido é revertido.



Código de exemplo

O código de exemplo a seguir mostra como você pode criar um orquestrador saga usando Step Functions. Para ver o código completo, consulte o [GitHub repositório](#) deste exemplo.

Definições de tarefa

```

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);

```

Definições de Step function e máquina de estado

```

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 repositório

Para uma implementação completa da arquitetura de amostra desse padrão, consulte o GitHub repositório em <https://github.com/aws-samples/saga-orchestration-netcore-blog>.

Referências do blog

- [Construir um aplicativo distribuído sem servidor usando o padrão Saga Orchestration](#)

Conteúdo relacionado

- [Padrão de coreografia saga](#)
- [Padrão de caixa de saída transacional](#)

Vídeos

O vídeo a seguir analisa como implementar o padrão de orquestração saga usando o AWS Step Functions.

Padrão scatter-gather

Intenção

O padrão scatter-gather é um padrão de roteamento de mensagens que envolve a transmissão de solicitações semelhantes ou relacionadas para vários destinatários e a agregação de suas respostas em uma única mensagem usando um componente chamado agregador. Esse padrão ajuda a alcançar a paralelização, reduz a latência de processamento e gerencia a comunicação assíncrona. É fácil implementar o padrão scatter-gather usando uma abordagem síncrona, mas uma abordagem mais robusta envolve implementá-lo como roteamento de mensagens em comunicação assíncrona, com ou sem um serviço de mensagens.

Motivação

No processamento de aplicações, uma solicitação que pode levar muito tempo para ser processada sequencialmente pode ser dividida em várias solicitações processadas paralelamente. Você também pode enviar solicitações para vários sistemas externos por meio de chamadas de API para obter uma resposta. O padrão scatter-gather é útil quando você precisa de informações de várias fontes. O Scatter-gather agrega os resultados para ajudar você a tomar uma decisão informada ou selecionar a melhor resposta para a solicitação.

O padrão scatter-gather consiste em duas fases, como o próprio nome indica:

- A fase de dispersão processa a mensagem de solicitação e a envia para vários destinatários em paralelo. Durante essa fase, a aplicação dispersa as solicitações pela rede e continua sendo executada sem esperar por respostas imediatas.
- Durante a fase de coleta, a aplicação coleta as respostas dos destinatários e as filtra ou combina em uma resposta unificada. Quando todas as respostas tiverem sido coletadas, elas poderão ser agregadas em uma única resposta, ou a melhor resposta poderá ser escolhida para processamento posterior.

Aplicabilidade

Use o padrão scatter-gather quando:

- Você planeja agregar e consolidar dados de vários APIs para criar uma resposta precisa. O padrão consolida informações de fontes diferentes em um todo coeso. Por exemplo, um sistema de agendamento pode fazer uma solicitação a vários destinatários para obter cotações de vários parceiros externos.
- A mesma solicitação precisa ser enviada para vários destinatários simultaneamente para concluir uma transação. Por exemplo, você pode usar esse padrão para consultar dados de inventário em paralelo e verificar a disponibilidade de um produto.
- Você deseja implementar um sistema confiável e escalável em que o balanceamento de carga possa ser obtido distribuindo solicitações entre vários destinatários. Se um destinatário falhar ou enfrentar uma carga alta, outros destinatários ainda poderão processar solicitações.
- Você quer otimizar a performance ao implementar consultas complexas que envolvem várias fontes de dados. Você pode dispersar a consulta em bancos de dados relevantes, reunir os resultados parciais e combiná-los em uma resposta abrangente.
- Você está implementando um tipo de processamento de redução de mapas em que a solicitação de dados é roteada para vários endpoints de processamento de dados para fragmentação e replicação. Os resultados parciais são filtrados e combinados para compor a resposta correta.
- Você deseja distribuir as operações de gravação em um espaço de chave de partição em workloads com gravação intensa em banco de dados de chave-valor. O agregador lê os resultados consultando os dados em cada fragmento e os consolida em uma única resposta.

Problemas e considerações

- Tolerância a falhas: esse padrão depende de vários destinatários que trabalham em paralelo, por isso é essencial lidar com as falhas normalmente. Para mitigar o impacto das falhas do destinatário no sistema geral, você pode implementar estratégias como redundância, replicação e detecção de falhas.
- Limites de escalabilidade: à medida que o número total de nós de processamento aumenta, a sobrecarga de rede associada também aumenta. Cada solicitação que envolve comunicação pela rede pode aumentar a latência e afetar negativamente os benefícios da paralelização.
- Gargalos de tempo de resposta: para operações que exigem que todos os destinatários sejam processados antes que o processamento final seja concluído, a performance geral do sistema é limitada pelo tempo de resposta mais lento do destinatário.
- Respostas parciais: quando as solicitações são dispersadas para vários destinatários, alguns destinatários podem atingir o tempo limite. Nesses casos, a implementação deve comunicar ao

cliente que a resposta está incompleta. Você também pode exibir os detalhes da agregação de respostas usando um frontend de interface de usuário.

- **Consistência de dados:** ao processar dados de vários destinatários, você deve considerar cuidadosamente as técnicas de sincronização de dados e resolução de conflitos, para garantir que os resultados agregados finais sejam precisos e consistentes.

Implementação

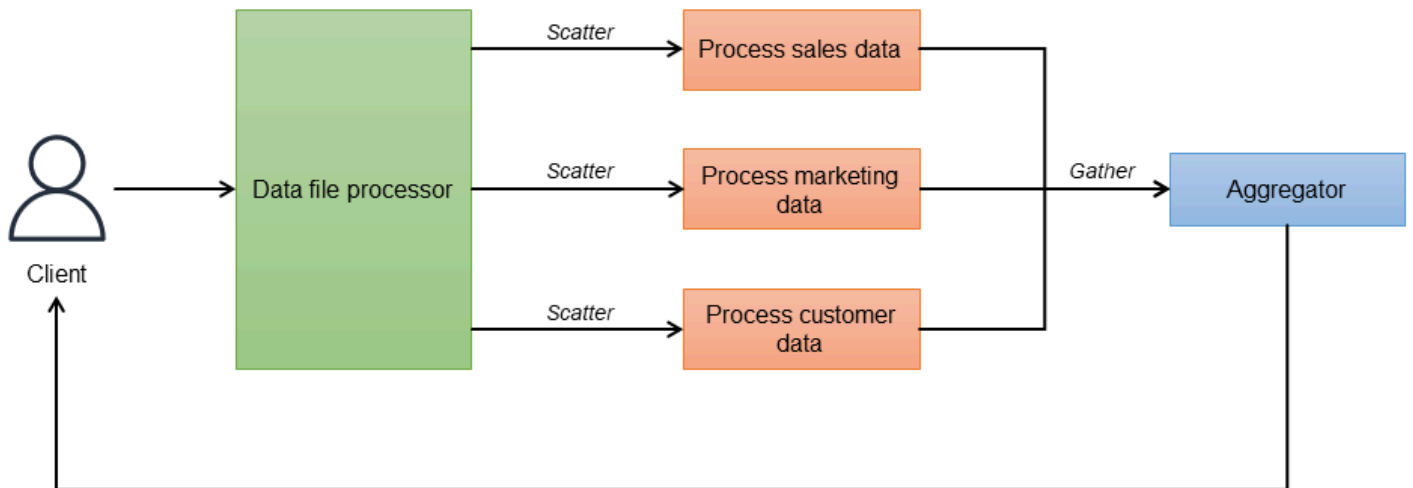
Arquitetura de alto nível

O padrão scatter-gather usa um controlador raiz para distribuir solicitações aos destinatários que processarão as solicitações. Durante a fase de dispersão, esse padrão pode usar dois mecanismos para enviar mensagens aos destinatários:

- **Dispersão por distribuição:** a aplicação tem uma lista conhecida de destinatários que devem ser chamados para obter os resultados. Os destinatários podem ser processos diferentes com funções exclusivas ou um único processo que teve a escala aumentada horizontalmente para distribuir a carga de processamento. Se algum dos nós de processamento atingir o tempo limite ou mostrar atrasos na resposta, o controlador poderá redistribuir o processamento para outro nó.
- **Dispersão por leilão:** a aplicação transmite a mensagem aos destinatários interessados usando um [padrão de publicação-assinatura](#). Nesse caso, os destinatários podem assinar a mensagem ou cancelar a assinatura a qualquer momento.

Dispersão por distribuição

No método de dispersão por distribuição, o controlador raiz divide a solicitação recebida em tarefas independentes e as atribui aos destinatários disponíveis (a fase de dispersão). Cada destinatário (processo, contêiner ou função do Lambda) trabalha de forma independente e paralela em sua computação e produz uma parte da resposta. Quando os destinatários concluem suas tarefas, eles enviam suas respostas para um agregador (a fase de coleta). O agregador combina as respostas parciais e retorna o resultado final ao cliente. O diagrama a seguir ilustra esse fluxo de trabalho.



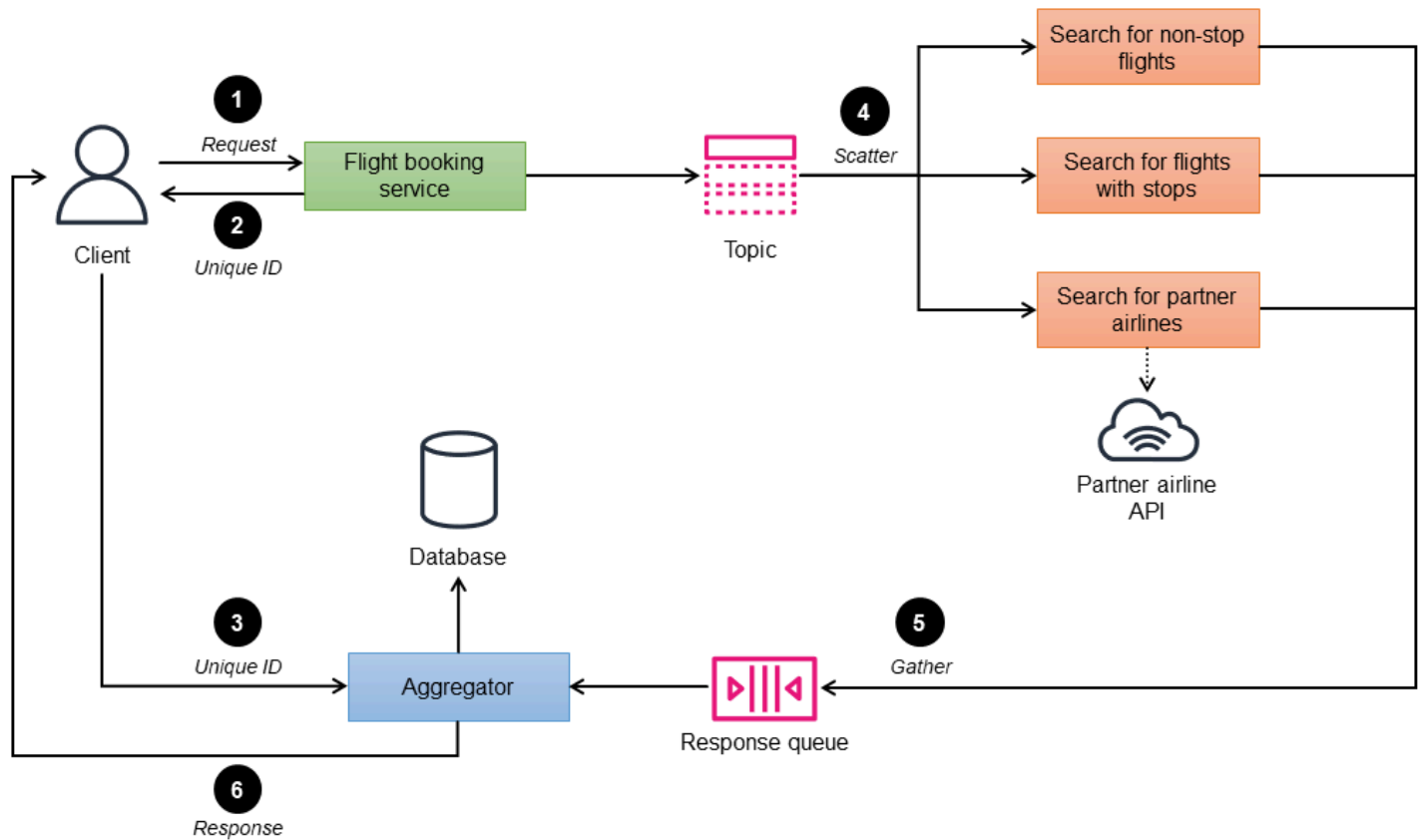
O controlador (processador de arquivos de dados) orquestra todo o conjunto de invocações e está ciente de todos os endpoints de reserva a serem chamados. Ele pode configurar um parâmetro de tempo limite para ignorar respostas que demoram muito. Quando as solicitações são enviadas, o agregador aguarda as respostas de cada endpoint. Para implementar a resiliência, cada microsserviço pode ser implantado com várias instâncias para balanceamento de carga. O agregador obtém os resultados, os combina em uma única mensagem de resposta e remove dados duplicados antes de continuar o processamento. As respostas que expiram são ignoradas. O controlador também pode atuar como um agregador em vez de usar um serviço agregador separado.

Dispersão por leilão

Se o controlador não estiver ciente dos destinatários, ou se os destinatários estiverem com acoplamento fraco, você poderá usar o método de dispersão por leilão. Nesse método, os destinatários se inscrevem em um tópico e o controlador publica a solicitação no tópico. Os destinatários publicam os resultados em uma fila de respostas. Como o controlador raiz não conhece os destinatários, o processo de coleta usa um agregador (outro padrão de mensagens) para coletar as respostas e transformá-las em uma única mensagem de resposta. O agregador usa um ID exclusivo para identificar um grupo de solicitações.

Por exemplo, no diagrama a seguir, o método de dispersão por leilão é usado para implementar um serviço de reserva de voos no site de uma companhia aérea. O site permite que os usuários pesquisem e exibam voos da própria companhia aérea e das companhias aéreas parceiras, e deve exibir o status da pesquisa em tempo real. O serviço de reserva de voos consiste em três microsserviços de busca: voos sem escala, voos com escalas e companhias aéreas parceiras.

A pesquisa da companhia aérea parceira chama os endpoints da API do parceiro para obter as respostas.

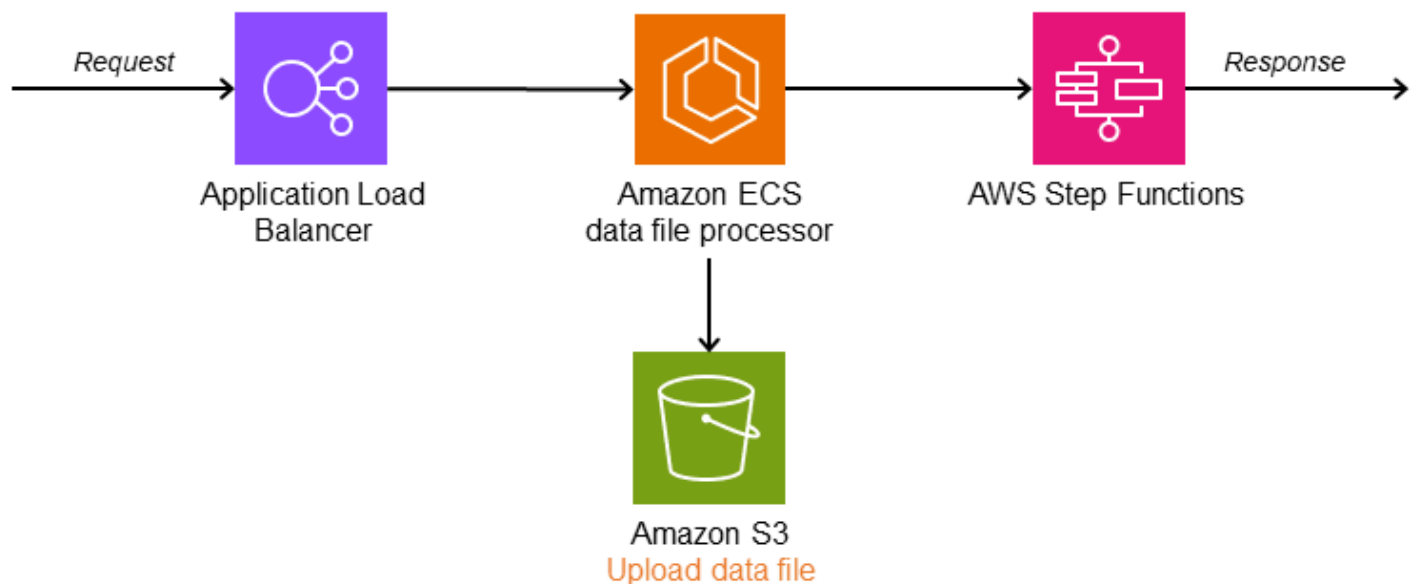


1. O serviço de reserva de voos (controlador) usa os critérios de pesquisa como entrada do cliente e processa e publica a solicitação no tópico.
2. O controlador usa uma ID exclusiva para identificar cada grupo de solicitações.
3. O cliente envia a ID exclusiva para o agregador na etapa 6.
4. Os microsserviços de pesquisa de reservas que se inscreveram no tópico de reserva recebem a solicitação.
5. Os microsserviços processam a solicitação e devolvem a disponibilidade de assentos para os critérios de pesquisa fornecidos em uma fila de resposta.
6. O agregador agrupa todas as mensagens de resposta armazenadas em um banco de dados temporário, agrupa os voos por ID exclusiva, cria uma única resposta unificada e a envia de volta ao cliente.

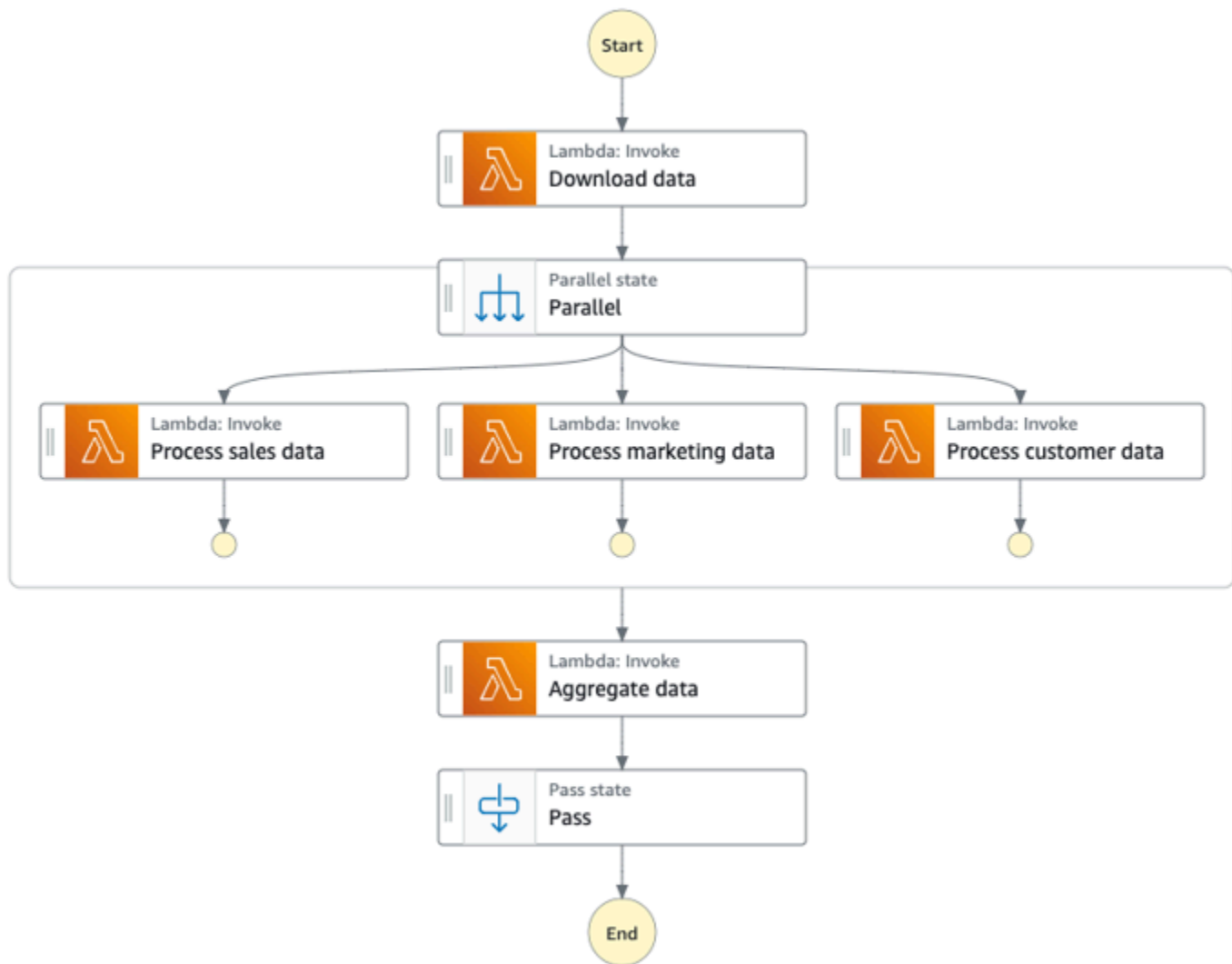
Implementação usando Serviços da AWS

Dispersão por distribuição

Na arquitetura a seguir, o controlador raiz é um processador de arquivos de dados (Amazon ECS) que divide os dados da solicitação recebida em buckets individuais do Amazon Simple Storage Service (Amazon S3) e inicia um fluxo de trabalho. AWS Step Functions baixa os dados e inicia o processamento paralelo dos arquivos. O estado `Parallel` espera que todas as tarefas retornem uma resposta. Uma AWS Lambda função agrega os dados e os salva de volta no Amazon S3.

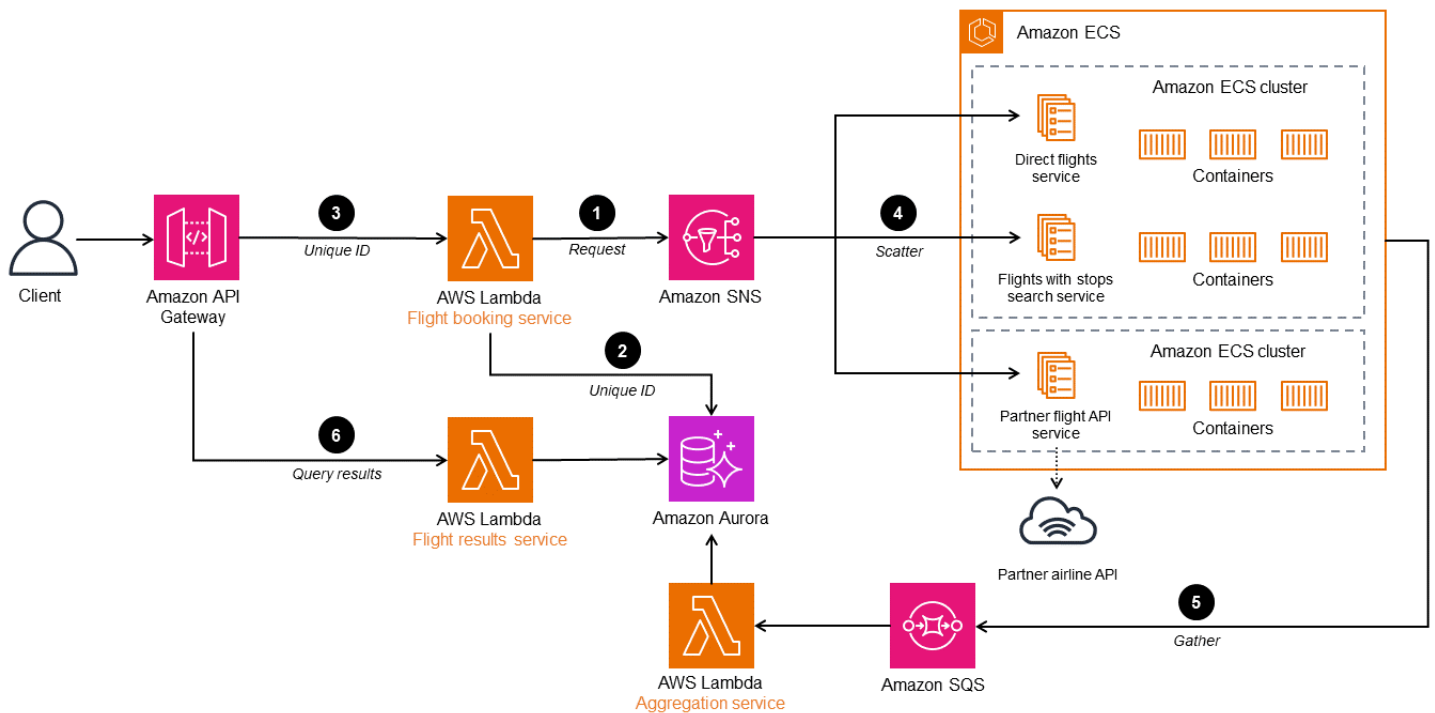


O diagrama a seguir ilustra o fluxo de trabalho do Step Functions com o estado `Parallel`.



Dispersão por leilão

O diagrama a seguir mostra uma AWS arquitetura para o método de dispersão por leilão. O serviço de reserva de voos do controlador raiz distribui a solicitação de pesquisa de voos em vários microsserviços. Um canal de publicação/assinatura é implementado com o Amazon Simple Notification Service (Amazon SNS), que é um serviço de mensagens gerenciado para comunicações. O Amazon SNS oferece suporte a mensagens entre aplicações de microsserviços desacoplados ou comunicações diretas com os usuários. Você pode implantar os microsserviços do destinatário no Amazon Elastic Kubernetes Service (Amazon EKS) ou no Amazon Elastic Container Service (Amazon ECS) para um melhor gerenciamento e escalabilidade. O serviço de resultados do voo retorna os resultados para o cliente. Ele pode ser implementado em AWS Lambda ou em outros serviços de orquestração de contêineres, como Amazon ECS ou Amazon EKS.



1. O serviço de reserva de voos (controlador) usa os critérios de pesquisa como entrada do cliente e processa e publica a solicitação no tópico do SNS.
2. O controlador publica o ID exclusivo em um banco de dados Amazon Aurora para identificar a solicitação.
3. O cliente envia a ID exclusiva para o cliente na etapa 6.
4. Os microsserviços de pesquisa de reservas que se inscreveram no tópico de reserva recebem a solicitação.
5. Os microsserviços processam a solicitação e devolvem a disponibilidade de assentos para os critérios de pesquisa fornecidos para uma fila de resposta no Amazon Simple Queue Service (Amazon SQS). O agregador agrupa todas as mensagens de resposta e as armazena em um banco de dados temporário.
6. O serviço de resultados de voos agrupa os voos por ID exclusivo, cria uma única resposta unificada e a envia de volta ao cliente.

Se você quiser adicionar outra pesquisa aérea a essa arquitetura, adicione um microsserviço que assina o tópico do SNS e publica na fila do SQS.

Resumindo, o padrão scatter-gather permite que os sistemas distribuídos alcancem uma paralelização eficiente, reduzam a latência e lidem perfeitamente com a comunicação assíncrona.

GitHub repositório

Para uma implementação completa da arquitetura de amostra desse padrão, consulte o GitHub repositório em <https://github.com/aws-samples/asynchronous-messaging-workshop/tree/master/code/lab-3>.

Workshop

- [Scatter-gather lab](#) no workshop Decoupled Microservices

Referências do blog

- [Application integration patterns for microservices](#)

Conteúdo relacionado

- Padrão [publicar/assinar](#)

padrão strangler fig

Intenção

O padrão strangler fig ajuda a migrar incrementalmente uma aplicação monolítica para uma arquitetura de microsserviços, com risco de transformação e interrupção dos negócios reduzidos.

Motivação

As aplicações monolíticas são desenvolvidas para fornecer a maior parte de suas funcionalidades em um único processo ou contêiner. O código está fortemente acoplado. Como resultado, as alterações na aplicação exigem um novo teste completo para evitar problemas de regressão. As mudanças não podem ser testadas isoladamente, o que afeta o tempo do ciclo. À medida que a aplicação é aprimorada com mais recursos, a alta complexidade pode levar a mais tempo gasto em manutenção, maior tempo de lançamento no mercado e, conseqüentemente, retardar a inovação do produto.

Quando a aplicação aumenta de tamanho, ela aumenta a carga cognitiva da equipe e pode ocasionar limites pouco claros de propriedade da equipe. Não é possível escalar recursos individuais com base na carga. Toda a aplicação precisa ser escalada para suportar picos de carga. À medida que os sistemas envelhecem, a tecnologia pode se tornar obsoleta, o que aumenta os custos de suporte. As aplicações monolíticas legadas seguem as práticas recomendadas que estavam disponíveis no momento do desenvolvimento e não foram projetadas para serem distribuídas.

Quando uma aplicação monolítica é migrada para uma arquitetura de microsserviços, ela pode ser dividida em componentes menores. Esses componentes podem ser escalados de forma independente, podem ser lançados de forma independente e podem ser de propriedade de equipes individuais. Isso resulta em uma maior velocidade de mudança, porque as alterações são localizadas e podem ser testadas e liberadas rapidamente. As mudanças têm um escopo de impacto menor porque os componentes têm acoplamento fraco e podem ser implantados individualmente.

Substituir completamente um monólito por uma aplicação de microsserviços reescrevendo ou refatorando o código é uma grande tarefa e um grande risco. Uma grande migração, em que o monólito é migrado em uma única operação, introduz risco de transformação e interrupção nos negócios. Enquanto a aplicação está sendo refatorada, é extremamente difícil ou mesmo impossível adicionar novos recursos.

Uma forma de resolver esse problema é usar o padrão fig pattern, introduzido por Martin Fowler. Esse padrão envolve migrar para microsserviços extraíndo gradualmente recursos e criando uma nova aplicação em torno do sistema existente. Os recursos do monólito são substituídos gradualmente por microsserviços, e os usuários da aplicação podem usar os recursos recém-migrados progressivamente. Quando todos os recursos são transferidos para o novo sistema, a aplicação monolítica pode ser desativada com segurança.

Aplicabilidade

Use o padrão strangler fig quando:

- Você deseja migrar gradualmente sua aplicação monolítica para uma arquitetura de microsserviços.
- Uma abordagem de migração do big bang é arriscada devido ao tamanho e à complexidade do monólito.
- A empresa quer adicionar novos recursos e não pode esperar que a transformação seja concluída.
- Os usuários finais devem ser minimamente afetados durante a transformação.

Problemas e considerações

- Acesso à base de código: para implementar o padrão strangler fig, você deve ter acesso à base de código da aplicação monolítica. À medida que os recursos são migrados do monólito, você precisará fazer pequenas alterações no código e implementar uma camada anticorrupção dentro do monólito para encaminhar chamadas para novos microsserviços. Você não pode interceptar chamadas sem acesso à base de código. O acesso à base de código também é essencial para redirecionar as solicitações recebidas. Talvez seja necessária alguma refatoração de código para que a camada de proxy possa interceptar as chamadas dos recursos migrados e roteá-las para microsserviços.
- Domínio pouco claro: a decomposição prematura de sistemas pode ser cara, especialmente quando o domínio não está claro e é possível errar os limites do serviço. O design orientado por domínio (DDD) é um mecanismo para entender o domínio, e o event storming é uma técnica para determinar os limites do domínio.
- Identificação de microsserviços: você pode usar o DDD como uma ferramenta essencial para identificar microsserviços. Para identificar microsserviços, procure as divisões naturais entre as classes de serviço. Muitos serviços terão seu próprio objeto de acesso a dados e se separarão

facilmente. Serviços que têm lógica de negócios relacionada e classes que têm poucas ou nenhuma dependência são bons candidatos para microsserviços. Você pode refatorar o código antes de quebrar o monólito para evitar um acoplamento forte. Você também deve considerar os requisitos de conformidade, o ritmo de lançamento, a localização geográfica das equipes, as necessidades de escalabilidade, as necessidades de tecnologia orientadas por casos de uso e a carga cognitiva das equipes.

- **Camada anticorrupção:** durante o processo de migração, quando os recursos dentro do monólito precisarem chamar os recursos que foram migrados como microsserviços, você deve implementar uma camada anticorrupção (ACL) que roteie cada chamada para o microsserviço apropriado. Para desacoplar e evitar alterações nos chamadores existentes dentro do monólito, a ACL funciona como um adaptador ou uma fachada que converte as chamadas na interface mais recente. Isso é discutido em detalhes na [seção Implementação](#) do padrão da ACL, anteriormente neste guia.
- **Falha na camada de proxy:** durante a migração, uma camada de proxy intercepta as solicitações que vão para a aplicação monolítica e as roteia para o sistema legado ou para o novo sistema. No entanto, essa camada de proxy pode se tornar um único ponto de falha ou um gargalo de performance.
- **Complexidade da aplicação:** monólitos grandes são os que mais se beneficiam do padrão strangler fig. Para aplicações pequenas, em que a complexidade da refatoração completa é baixa, talvez seja mais eficiente reescrever a aplicação na arquitetura de microsserviços em vez de migrá-la.
- **Interações de serviço:** os microsserviços podem se comunicar de forma síncrona ou assíncrona. Quando a comunicação síncrona for necessária, considere se os tempos limite podem causar o consumo da conexão ou do grupo de threads, resultando em problemas de performance da aplicação. Nesses casos, use o [padrão do disjuntor](#) para retornar a falha imediata de operações que provavelmente falharão por longos períodos de tempo. A comunicação assíncrona pode ser obtida usando eventos e filas de mensagens.
- **Agregação de dados:** em uma arquitetura de microsserviços, os dados são distribuídos entre bancos de dados. Quando a agregação de dados é necessária, você pode usar o [AWS AppSync](#) no frontend ou o padrão de segmentação de responsabilidade de consulta de comando (CQRS) no backend.
- **Consistência de dados:** os microsserviços possuem seu armazenamento de dados, e a aplicação monolítica também pode provavelmente usar esses dados. Para permitir o compartilhamento, você pode sincronizar o armazenamento de dados dos novos microsserviços com o banco de dados da aplicação monolítica usando uma fila e um agente. No entanto, isso pode causar redundância de dados e eventual consistência entre dois armazenamentos de dados, por isso recomendamos

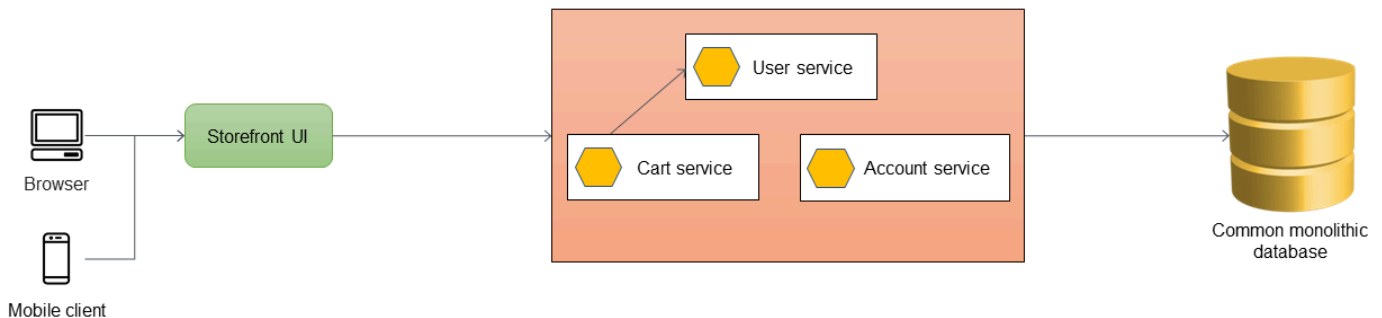
que você a trate como uma solução tática até que você possa estabelecer uma solução de longo prazo, como um data lake.

Implementação

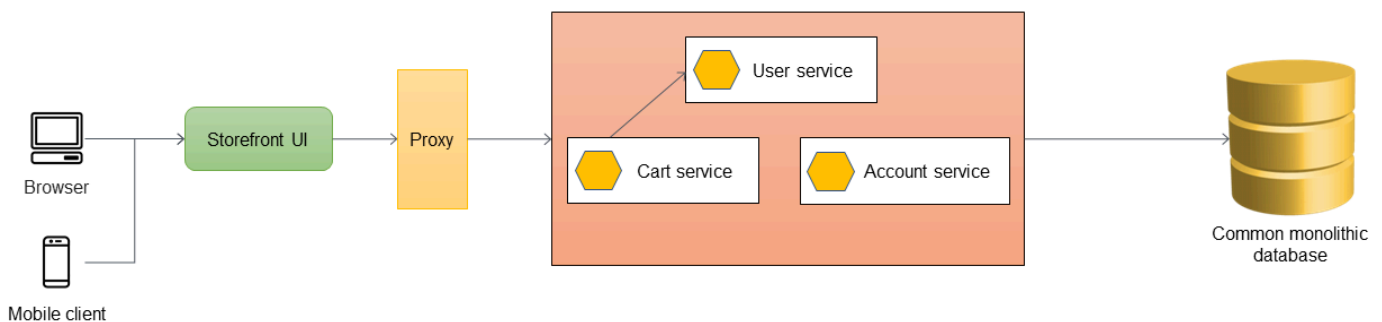
No padrão strangler fig, você substitui uma funcionalidade específica por um novo serviço ou aplicação, um componente por vez. Uma camada de proxy intercepta as solicitações que vão para a aplicação monolítica e as roteia para o sistema legado ou para o novo sistema. Como a camada de proxy roteia os usuários para a aplicação correta, você pode adicionar recursos ao novo sistema e, ao mesmo tempo, garantir que o monólito continue funcionando. O novo sistema eventualmente substitui todos os recursos do sistema antigo, e você pode desativá-lo.

Arquitetura de alto nível

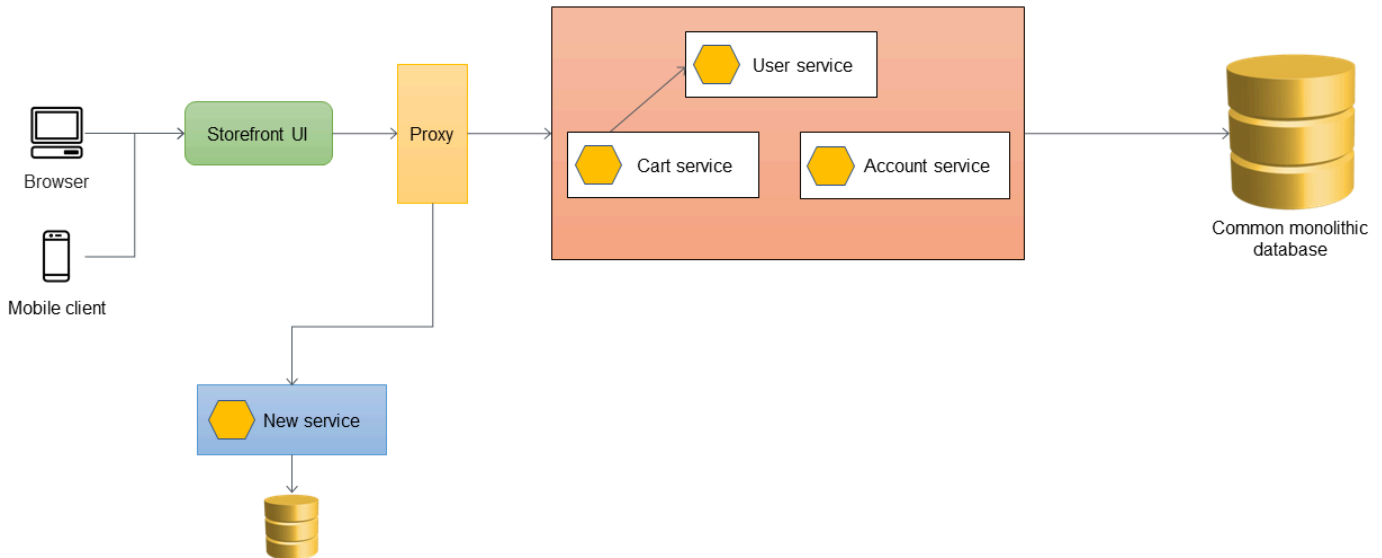
No diagrama a seguir, uma aplicação monolítica tem três serviços: serviço de usuário, serviço de carrinho e serviço de conta. O serviço de carrinho depende do serviço do usuário, e a aplicação usa um banco de dados relacional monolítico.



A primeira etapa é adicionar uma camada de proxy entre a interface do usuário do storefront e a aplicação monolítica. No início, o proxy roteia todo o tráfego para a aplicação monolítica.

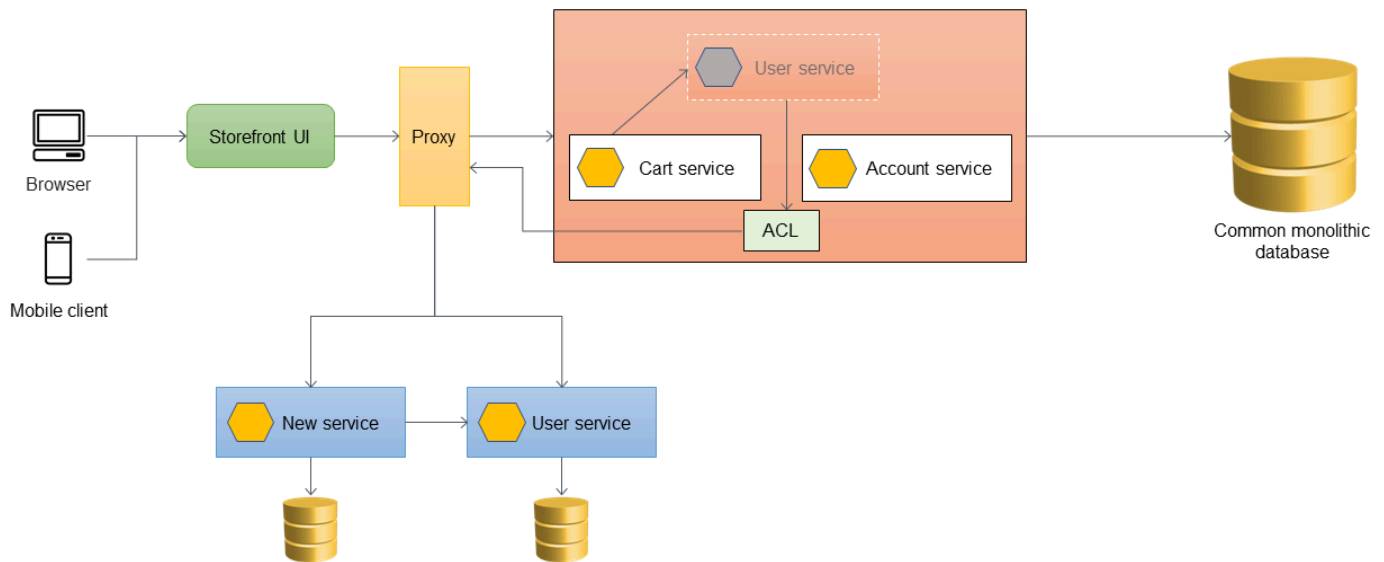


Quando quiser adicionar novos recursos à sua aplicação, você os implementa como novos microsserviços em vez de adicionar recursos ao monólito existente. No entanto, você continua corrigindo bugs no monólito para garantir a estabilidade da aplicação. No diagrama a seguir, a camada de proxy roteia as chamadas para o monólito ou para o novo microsserviço com base no URL da API.



Adição de uma camada anticorrupção

Na arquitetura a seguir, o serviço do usuário foi migrado para um microsserviço. O serviço de carrinho chama o serviço de usuário, mas a implementação não está mais disponível no monólito. Além disso, a interface do serviço recém-migrado pode não corresponder à interface anterior dentro da aplicação monolítica. Para lidar com essas alterações, você implementa uma ACL. Durante o processo de migração, quando os recursos do monólito precisam chamar os recursos que foram migrados como microsserviços, a ACL converte as chamadas na nova interface e as roteia para o microsserviço apropriado.

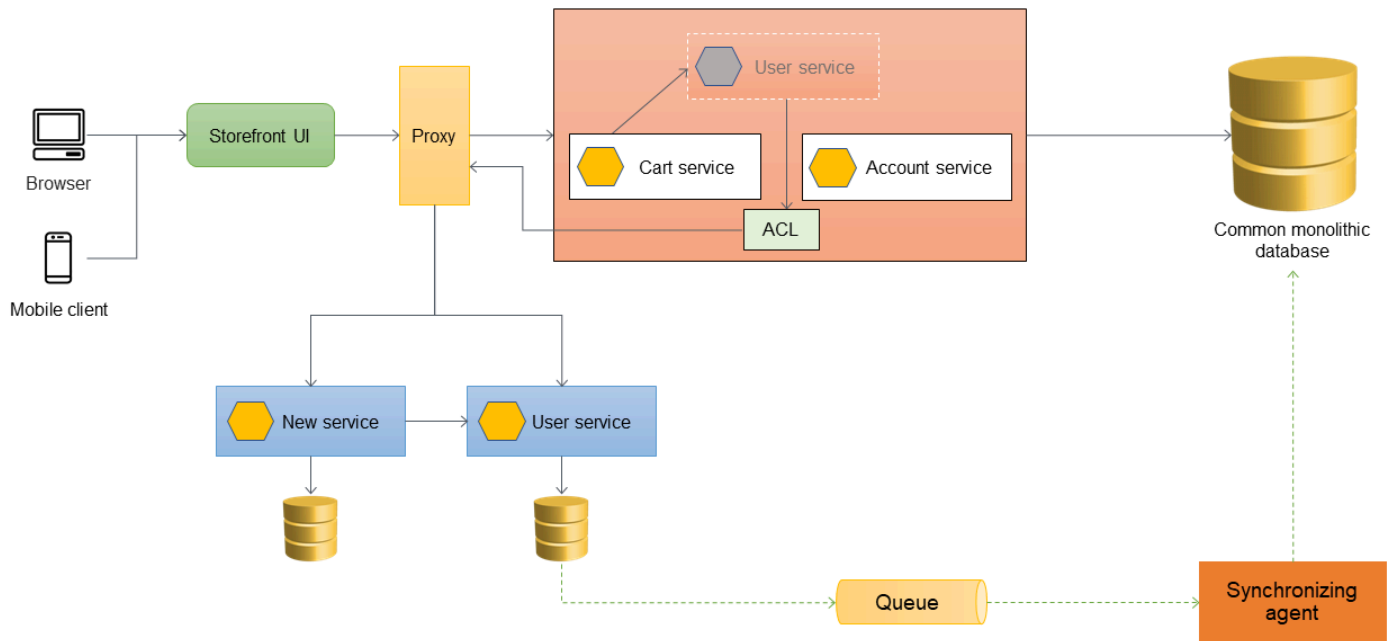


Você pode implementar a ACL dentro da aplicação monolítica como uma classe específica do serviço que foi migrado, por exemplo, `UserServiceFacade` ou `UserServiceAdapter`. A ACL deve ser desativada após a migração de todos os serviços dependentes para a arquitetura de microserviços.

Quando você usa a ACL, o serviço de carrinho ainda chama o serviço do usuário dentro do monólito, e o serviço do usuário redireciona a chamada para o microserviço por meio da ACL. O serviço de carrinho ainda deve chamar o serviço do usuário sem estar ciente da migração do microserviço. Esse acoplamento fraco é necessário para reduzir a regressão e a interrupção dos negócios.

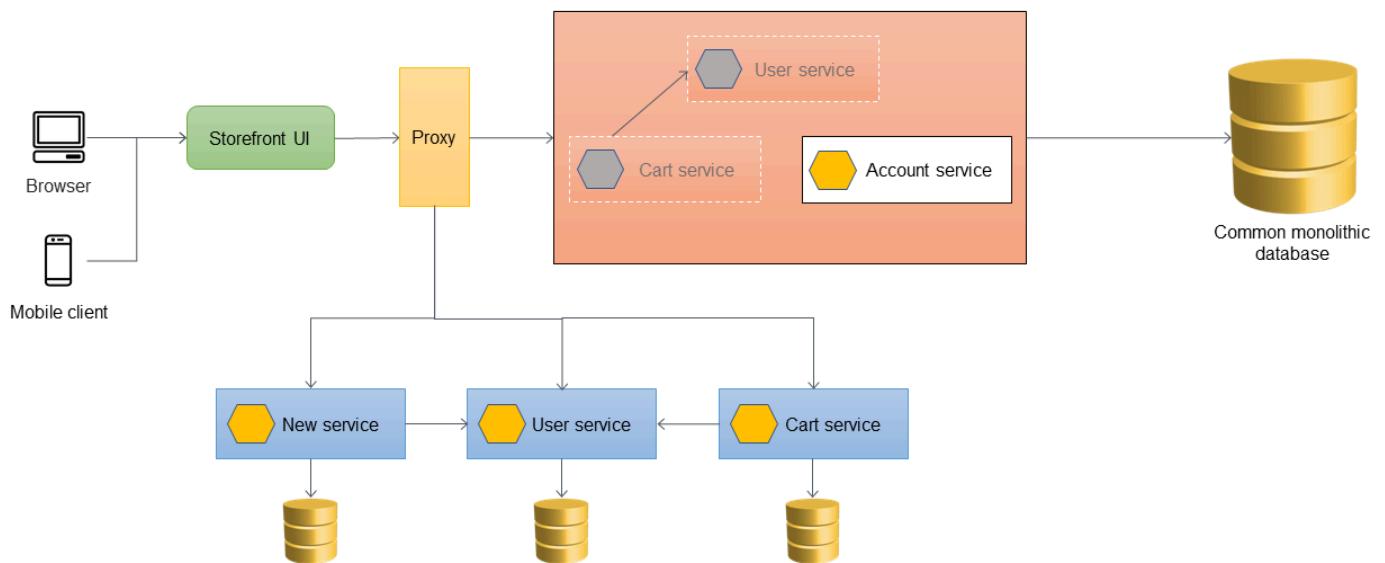
Gerenciamento da sincronização de dados

Como prática recomendada, o microserviço deve possuir os dados. O serviço do usuário armazena seus dados em seu próprio armazenamento de dados. Talvez seja necessário sincronizar dados com o banco de dados monolítico para lidar com dependências, como relatórios, e oferecer suporte a aplicações downstream que ainda não estão prontas para acessar diretamente os microserviços. A aplicação monolítica também pode exigir os dados de outras funções e componentes que ainda não foram migrados para microserviços. Portanto, a sincronização de dados é necessária entre o novo microserviço e o monólito. Para sincronizar os dados, você pode introduzir um agente de sincronização entre o microserviço do usuário e o banco de dados monolítico, conforme mostrado no diagrama a seguir. O microserviço do usuário envia um evento para a fila sempre que seu banco de dados é atualizado. O agente de sincronização escuta a fila e atualiza continuamente o banco de dados monolítico. Os dados no banco de dados monolítico acabam sendo consistentes com os dados que estão sendo sincronizados.



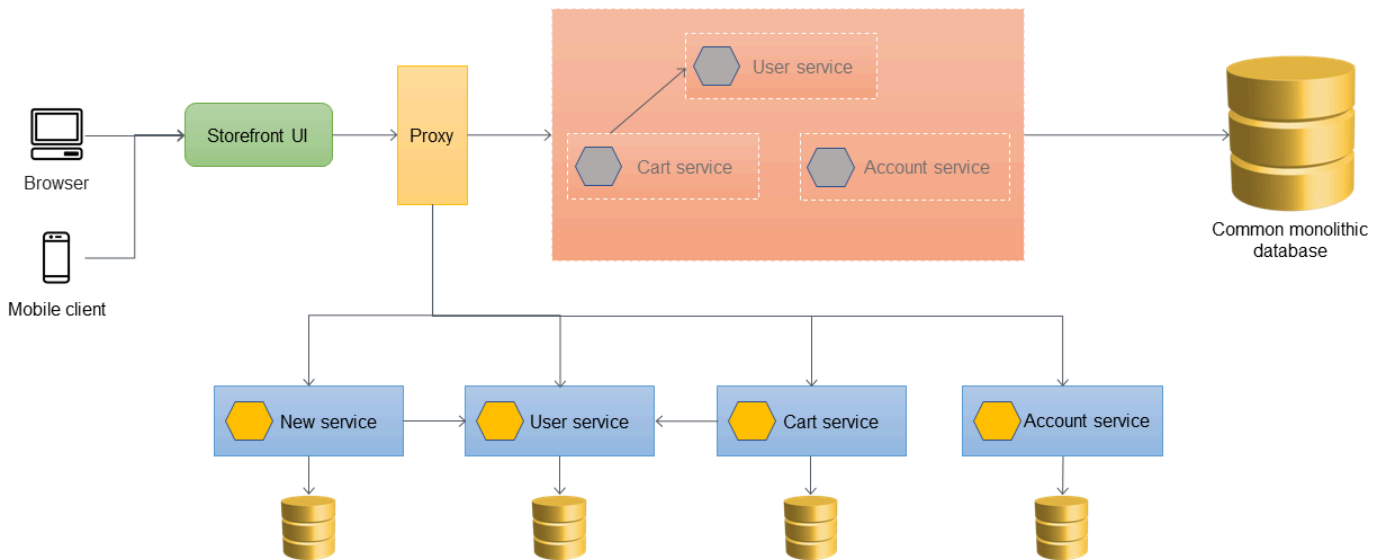
Migração de serviços adicionais

Quando o serviço de carrinho é migrado da aplicação monolítica, seu código é revisado para chamar o novo serviço diretamente, de forma que a ACL não roteie mais essas chamadas. O diagrama a seguir ilustra esse cenário

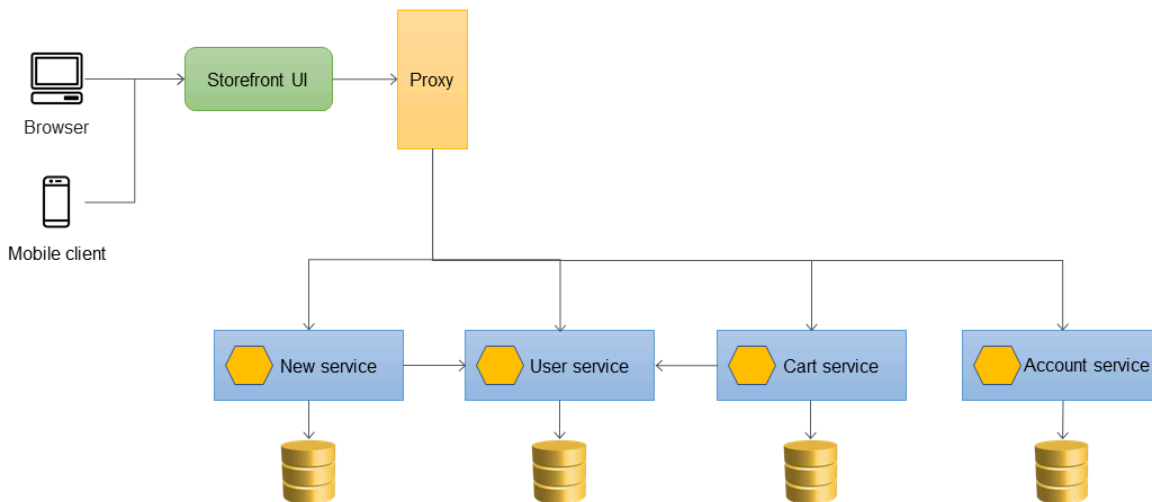


O diagrama a seguir mostra o estado final do processo de estrangulamento, em que todos os serviços foram migrados para fora do monólito, restando apenas o seu esqueleto. Os dados

históricos podem ser migrados para armazenamentos de dados pertencentes a serviços individuais. A ACL pode ser removida e o monólito está pronto para ser desativado neste estágio.



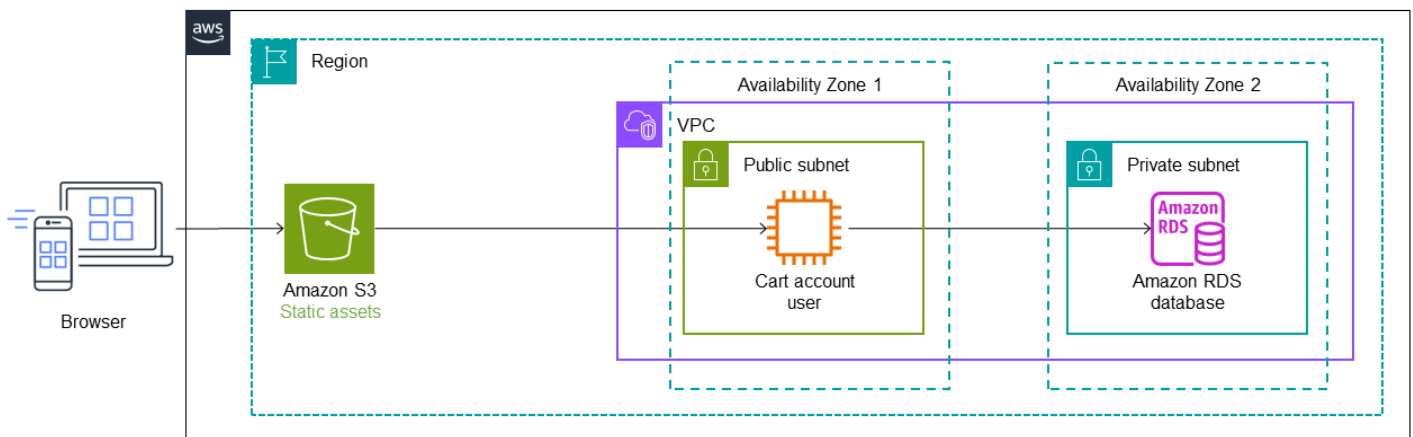
O diagrama a seguir mostra a arquitetura final após a desativação da aplicação monolítica. Você pode hospedar os microsserviços individuais por meio de um URL baseado em recursos (como `http://www.storefront.com/user`) ou por meio de seu próprio domínio (por exemplo, `http://user.storefront.com`) com base nos requisitos da sua aplicação. Para obter mais informações sobre os principais métodos para expor o HTTP APIs aos consumidores upstream usando nomes de host e caminhos, consulte a seção Padrões de [roteamento da API](#).



Implementação usando AWS serviços

Uso do API Gateway como proxy da aplicação

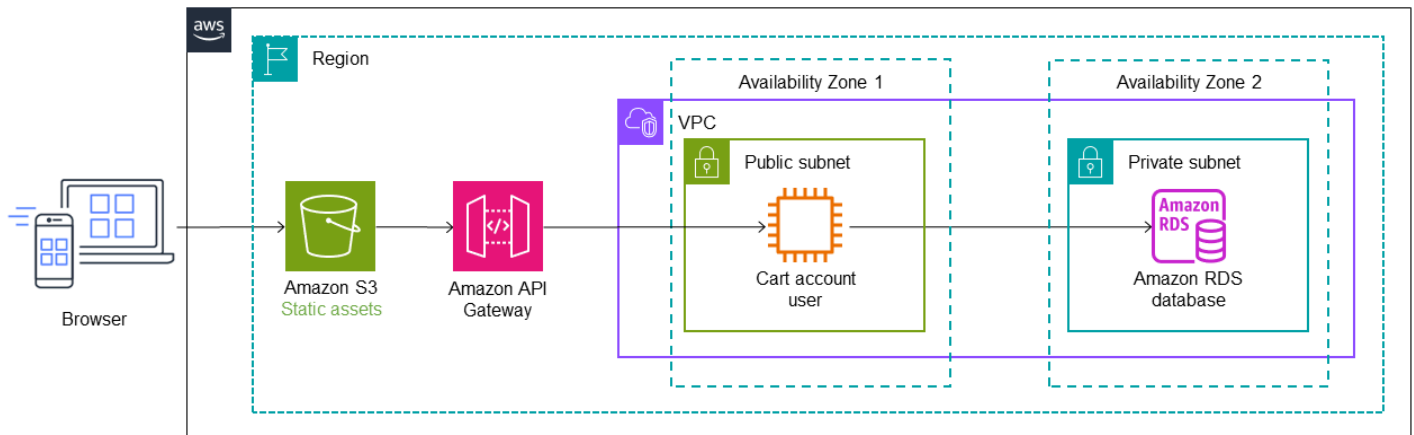
O diagrama a seguir mostra o estado inicial da aplicação monolítica. Vamos supor que ele foi migrado usando uma lift-and-shift estratégia, então está sendo executado em uma instância do [Amazon Elastic Compute Cloud \(Amazon EC2\)](#) e usa um banco de dados do Amazon Relational Database Service ([Amazon RDS](#)). Para simplificar, a arquitetura usa uma única nuvem privada virtual (VPC) com uma sub-rede privada e uma pública, e vamos supor que os microsserviços serão inicialmente implantados na mesma Conta da AWS. (A melhor prática em ambientes de produção é usar uma arquitetura de várias contas para garantir a independência da implantação.) A instância do EC2 reside em uma única zona de disponibilidade na sub-rede pública, e a instância do RDS reside em uma única zona de disponibilidade na sub-rede privada. [O Amazon Simple Storage Service \(Amazon S3\)](#) armazena ativos estáticos, como arquivos CSS e React, para JavaScript o site.



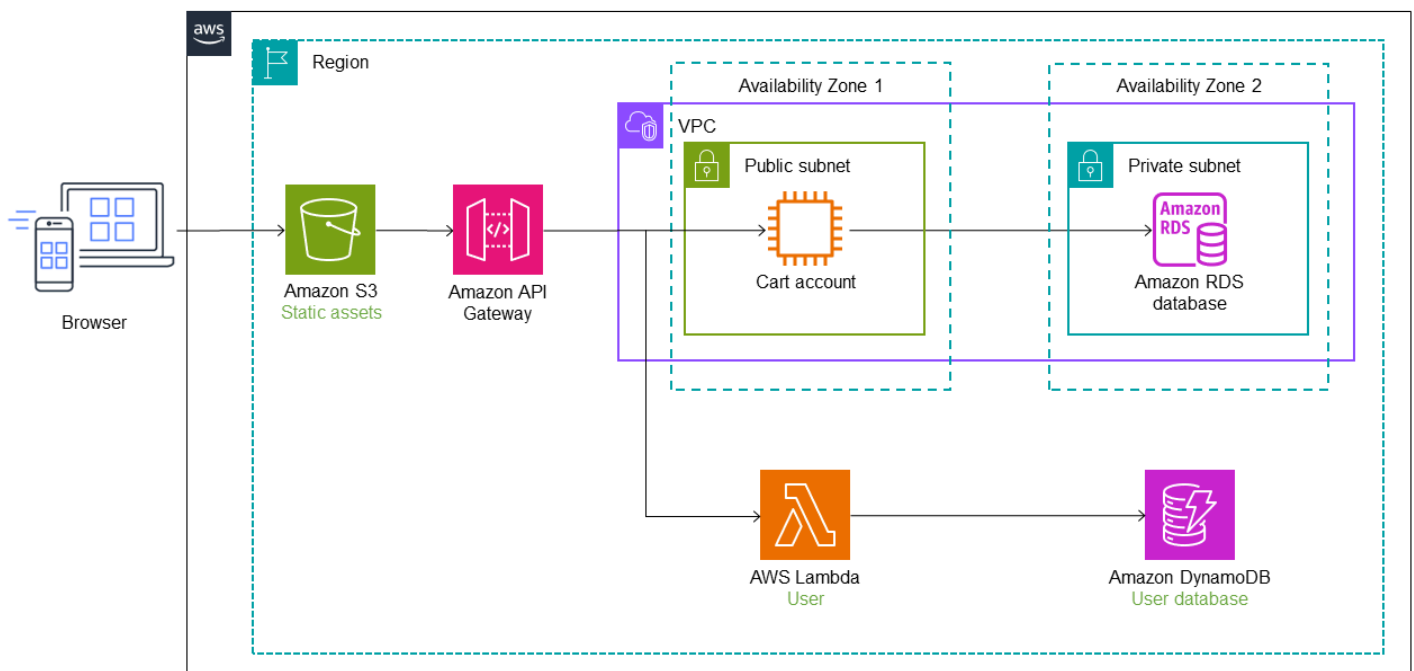
Na arquitetura a seguir, o [AWS Migration Hub Refactor Spaces](#) implanta o [Amazon API Gateway](#) na frente da aplicação monolítica. O Refactor Spaces cria uma infraestrutura de refatoração dentro da sua conta, e o API Gateway atua como a camada de proxy para rotear chamadas para o monólito. Inicialmente, todas as chamadas são roteadas para a aplicação monolítica por meio da camada de proxy. Conforme analisado anteriormente, as camadas de proxy podem se tornar um único ponto de falha. No entanto, usar o API Gateway como proxy reduz o risco porque é um serviço multi-AZ sem servidor.

Note

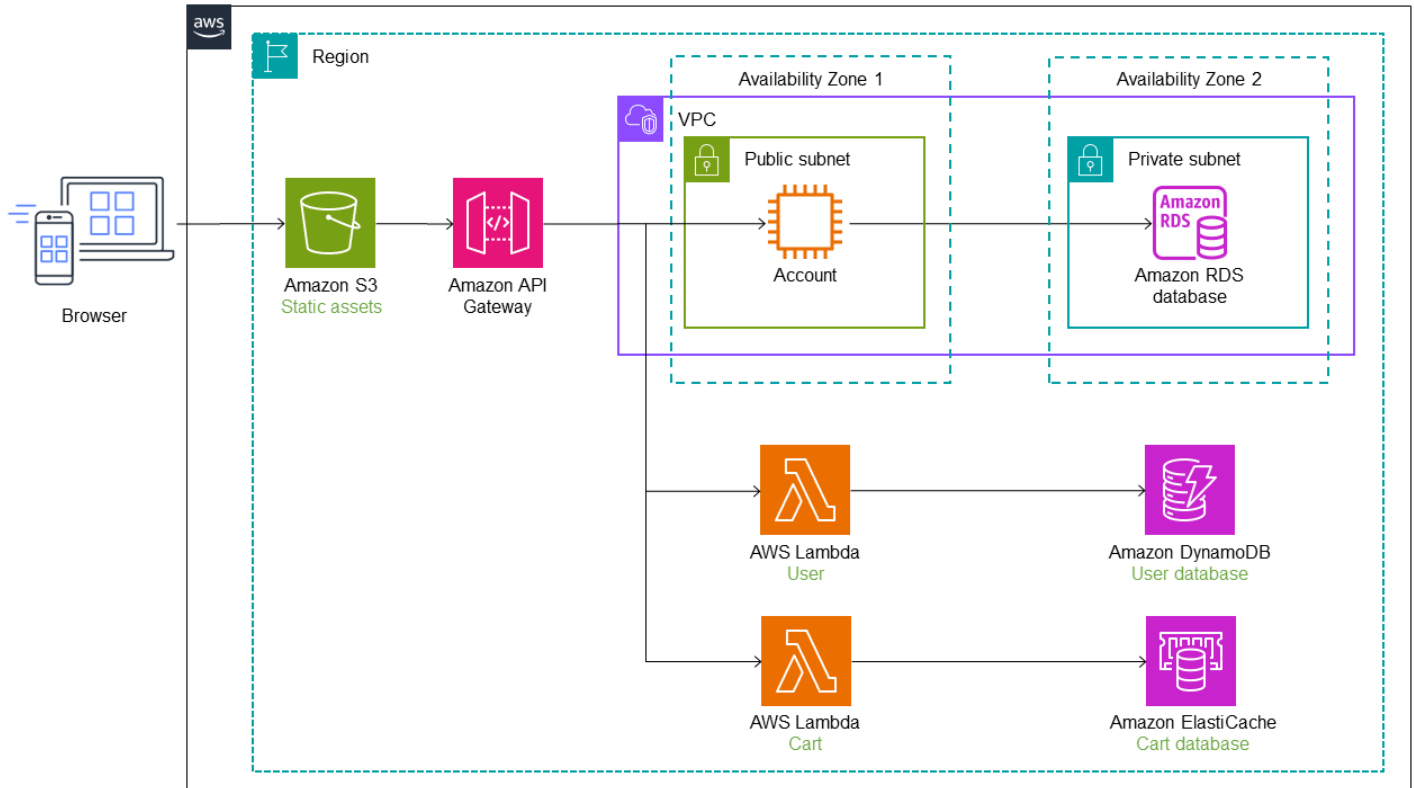
AWS Migration Hub Refactor Spaces não está mais aberto a novos clientes a partir de 7 de novembro de 2025. Para recursos semelhantes a AWS Migration Hub Refactor Spaces, explore [AWS Transform](#).



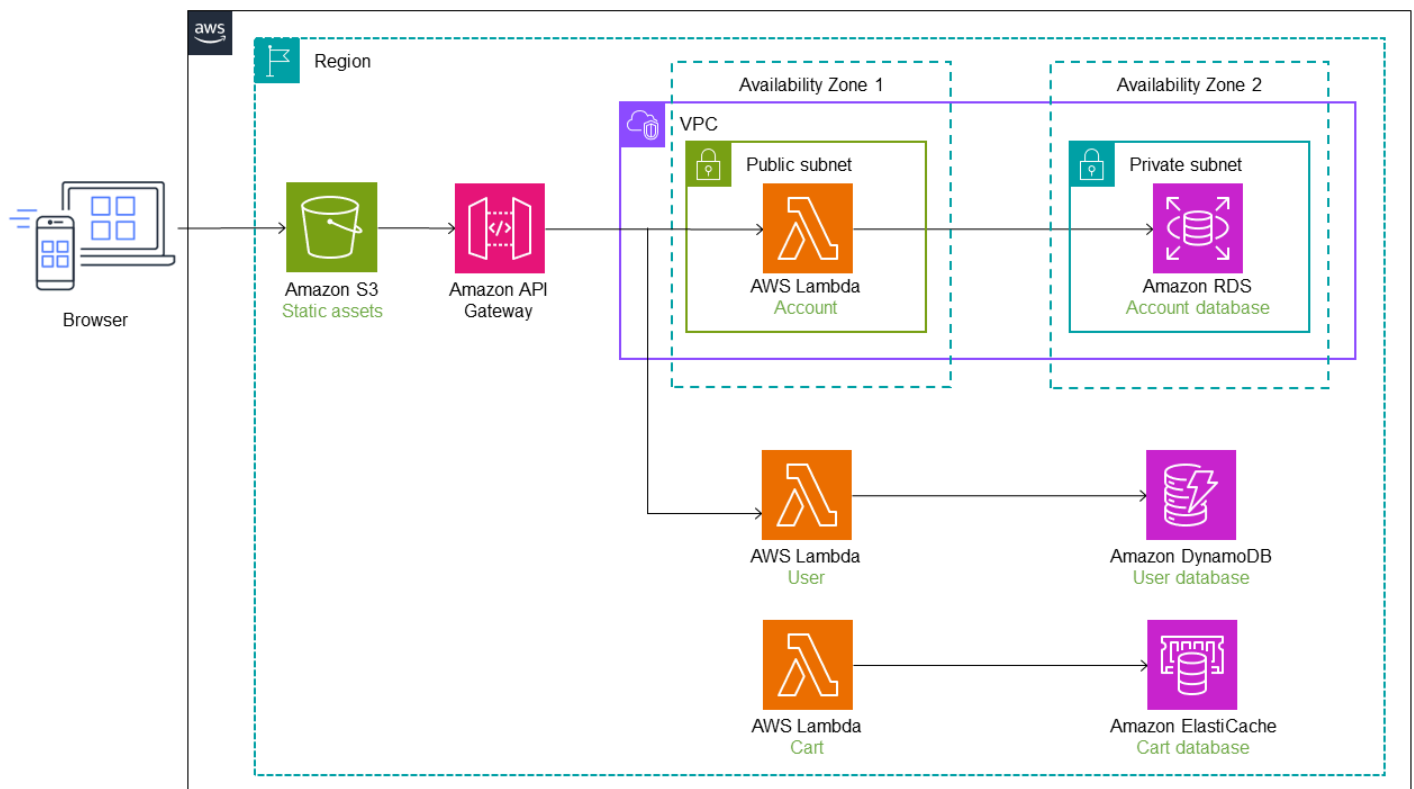
O serviço do usuário é migrado para uma função do Lambda, e um banco de dados do [Amazon DynamoDB](#) armazena seus dados. Um endpoint de serviço do Lambda e uma rota padrão são adicionados ao Refactor Spaces, e o API Gateway é configurado automaticamente para rotear as chamadas para a função do Lambda.



No diagrama a seguir, o serviço de carrinho também foi migrado do monólito para uma função do Lambda. Uma rota adicional e um endpoint de serviço são adicionados ao Refactor Spaces, e o tráfego passa automaticamente para a função Cart do Lambda. [O armazenamento de dados da função Lambda é gerenciado pela Amazon. ElastiCache](#) A aplicação monolítica ainda permanece na instância do EC2 junto com o banco de dados do Amazon RDS.



No diagrama a seguir, o último serviço (conta) é migrado do monólito para uma função do Lambda. Ele continua usando o banco de dados original do Amazon RDS. A nova arquitetura agora tem três microsserviços com bancos de dados separados. Cada serviço usa um tipo diferente de banco de dados. Esse conceito de usar banco de dados com propósito específico para atender às necessidades específicas dos microsserviços é denominado persistência poliglota. As funções do Lambda também podem ser implementadas em diferentes linguagens de programação, conforme determinado pelo caso de uso. Durante a refatoração, o Refactor Spaces automatiza a substituição e o roteamento do tráfego para o Lambda. Isso economiza para seus criadores o tempo necessário para arquitetar, implantar e configurar a infraestrutura de roteamento.



Uso de várias contas

Na implementação anterior, usamos uma única VPC com uma sub-rede pública e uma privada para a aplicação monolítica, e implantamos os microsserviços dentro da mesma Conta da AWS por uma questão de simplicidade. No entanto, esse raramente é o caso em cenários do mundo real, em que os microsserviços geralmente são implantados em várias Contas da AWS para independência de implantação. Em uma estrutura de várias contas, você precisa configurar o tráfego de roteamento do monólito para os novos serviços em contas diferentes.

O [Refactor Spaces](#) ajuda você a criar e configurar a AWS infraestrutura para rotear chamadas de API fora do aplicativo monolítico. O Refactor Spaces orquestra o [API Gateway](#), o [Network Load Balancer](#) e as políticas do [AWS Identity and Access Management \(IAM\)](#) baseadas em recursos nas suas contas da AWS como parte de seu recurso de aplicação. Você pode adicionar novos serviços de forma transparente em uma única conta Conta da AWS ou em várias contas a um endpoint HTTP externo. Todos esses recursos são orquestrados dentro do seu Conta da AWS e podem ser personalizados e configurados após a implantação.

Suponhamos que os serviços de usuário e carrinho de compras sejam implantados em duas contas diferentes, conforme mostrado no diagrama a seguir. Ao usar o Refactor Spaces, você só precisa configurar o endpoint do serviço e a rota. O Refactor Spaces automatiza a integração [API Gateway](#)–

Padrão de caixa de saída transacional

Intenção

O padrão de caixa de saída transacional resolve o problema de operações de gravação dupla que ocorre em sistemas distribuídos quando uma única operação envolve uma operação de gravação no banco de dados e uma notificação de mensagem ou evento. Uma operação de gravação dupla ocorre quando um aplicativo grava em dois sistemas diferentes; por exemplo, quando um microsserviço precisa manter os dados no banco de dados e enviar uma mensagem para notificar outros sistemas. Uma falha em uma dessas operações pode resultar em dados inconsistentes.

Motivação

Quando um microsserviço envia uma notificação de evento após uma atualização do banco de dados, essas duas operações devem ser executadas atomicamente para garantir a consistência e a confiabilidade dos dados.

- Se a atualização do banco de dados for bem-sucedida, mas a notificação do evento falhar, o serviço seguinte não estará ciente da alteração e o sistema poderá entrar em um estado inconsistente.
- Se a atualização do banco de dados falhar, mas a notificação do evento for enviada, os dados poderão ser corrompidos, o que poderá afetar a confiabilidade do sistema.

Aplicabilidade

Use o padrão de caixa de saída transacional quando:

- Você estiver criando um aplicativo orientado por eventos em que uma atualização do banco de dados inicia uma notificação de evento.
- Você quiser garantir a atomicidade em operações que envolvem dois serviços.
- Você quiser implementar o [padrão de fornecimento de eventos](#).

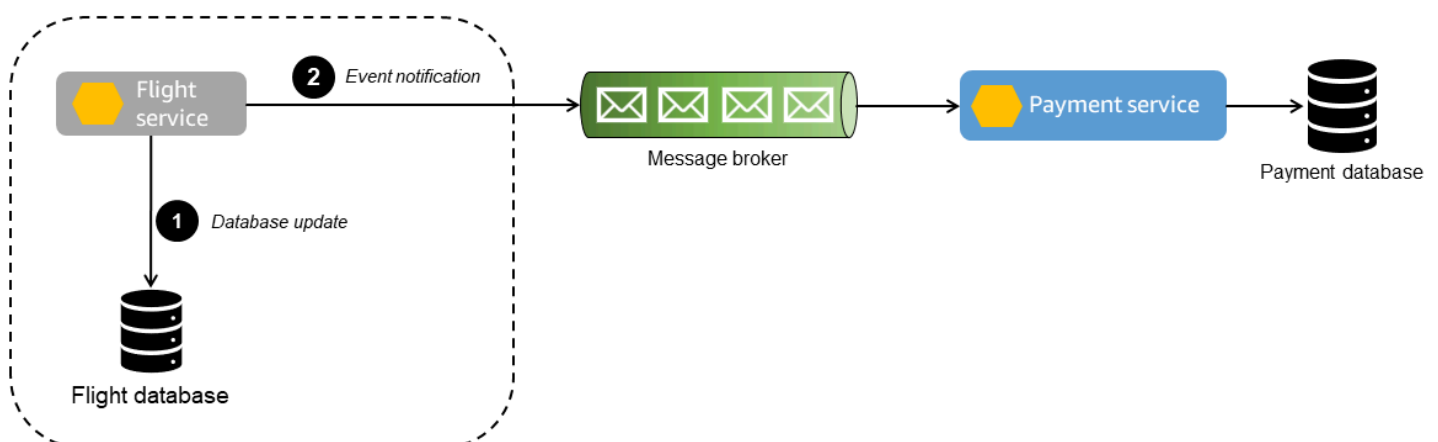
Problemas e considerações

- Mensagens duplicadas: o serviço de processamento de eventos pode enviar mensagens ou eventos duplicados, portanto, recomendamos que você torne o serviço consumidor idempotente rastreando as mensagens processadas.
- Ordem de notificação: envie mensagens ou eventos na mesma ordem em que o serviço atualiza o banco de dados. Isso é fundamental para o padrão de fornecimento de eventos, em que você pode usar um armazenamento de eventos para point-in-time recuperação do armazenamento de dados. Se a ordem estiver incorreta, isso poderá comprometer a qualidade dos dados. A consistência eventual e a reversão do banco de dados podem agravar o problema se a ordem das notificações não for preservada.
- Reversão da transação: não envie uma notificação de evento se a transação for revertida.
- Tratamento de transações em nível de serviço: se a transação abranger serviços que exigem atualizações do repositório de dados, use o [padrão de orquestração da saga](#) para preservar a integridade dos dados nos repositórios de dados.

Implementação

Arquitetura de alto nível

O diagrama de sequência a seguir mostra a ordem dos eventos que acontecem durante as operações de gravação dupla.



1. O serviço de voo grava no banco de dados e envia uma notificação de evento para o serviço de pagamento.

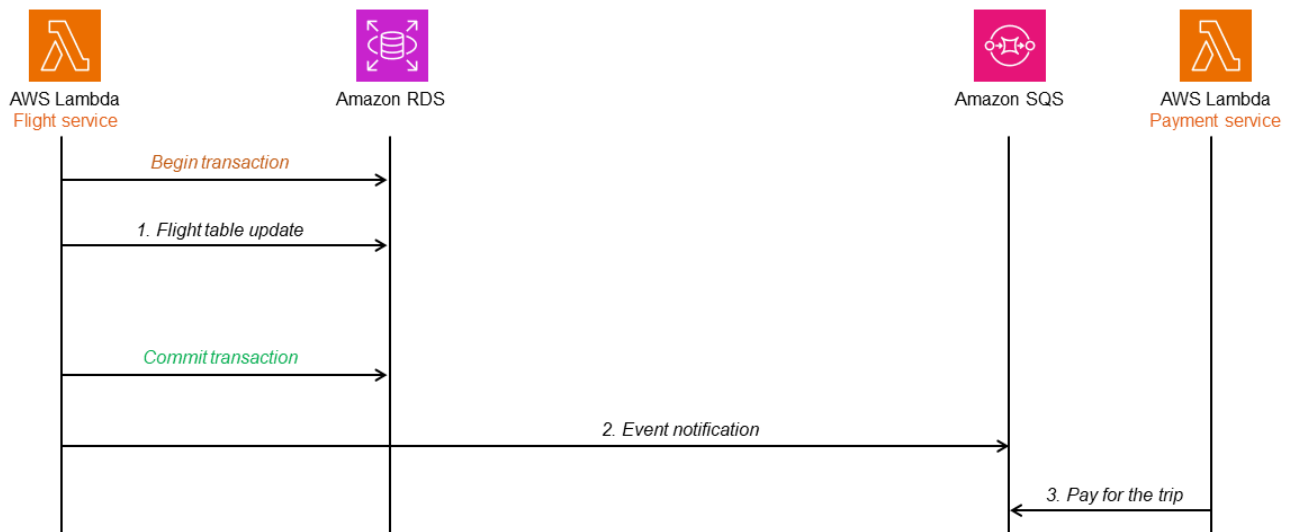
2. O agente de mensagens transporta as mensagens e os eventos para o serviço de pagamento. Qualquer falha no agente de mensagens impede que o serviço de pagamento receba as atualizações.

Se a atualização do banco de dados de voos falhar, mas a notificação for enviada, o serviço de pagamento processará o pagamento com base na notificação do evento. Isso causará inconsistências de dados posteriores.

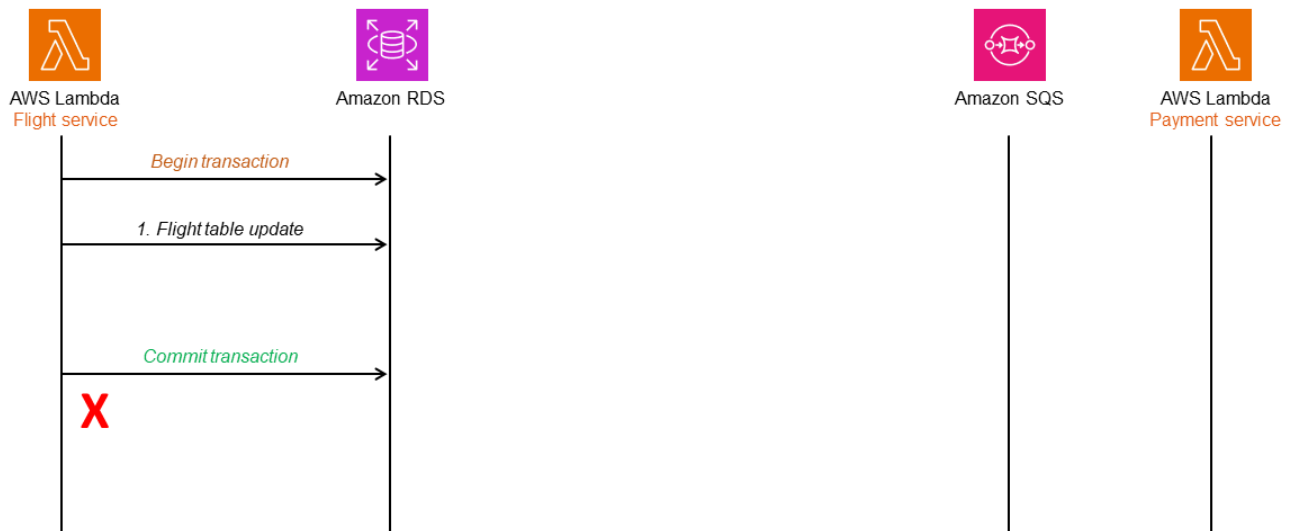
Implementação usando AWS serviços

Para demonstrar o padrão no diagrama de sequência, usaremos os serviços abaixo da AWS , conforme mostrado no diagrama a seguir.

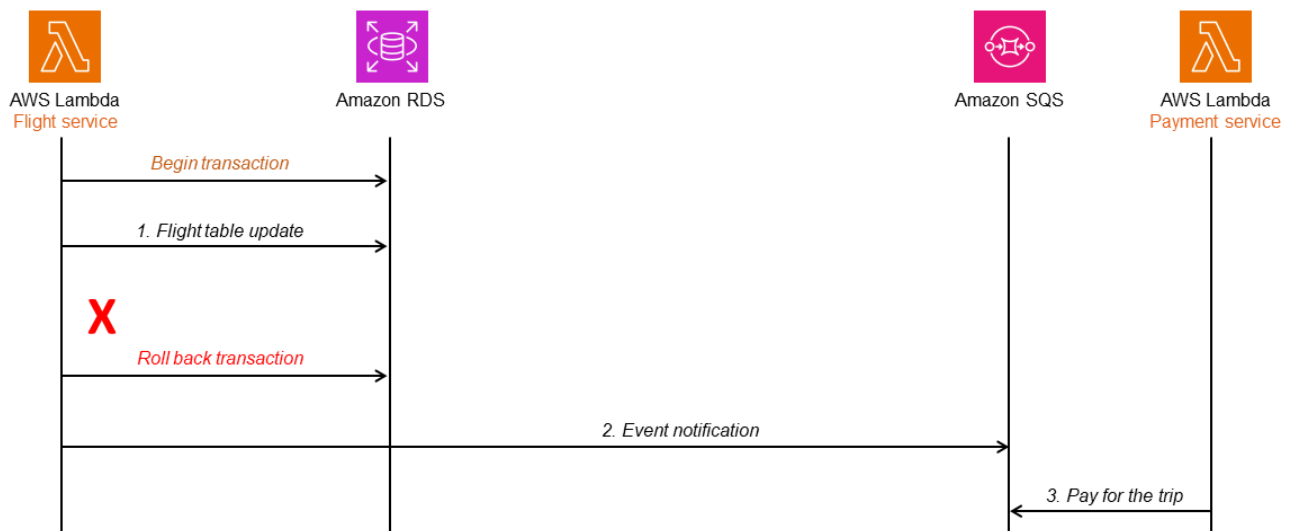
- Os microsserviços são implementados usando [AWS Lambda](#).
- O banco de dados primário é gerenciado pelo [Amazon Relational Database Service \(Amazon RDS\)](#).
- O [Amazon Simple Queue Service \(Amazon SQS\)](#) atua como o agente de mensagens que recebe notificações de eventos.



Se o serviço de voo falhar após a confirmação da transação, isso poderá fazer com que a notificação do evento não seja enviada.



No entanto, a transação pode falhar e ser revertida, mas a notificação do evento ainda pode ser enviada, fazendo com que o serviço de pagamento processe o pagamento.



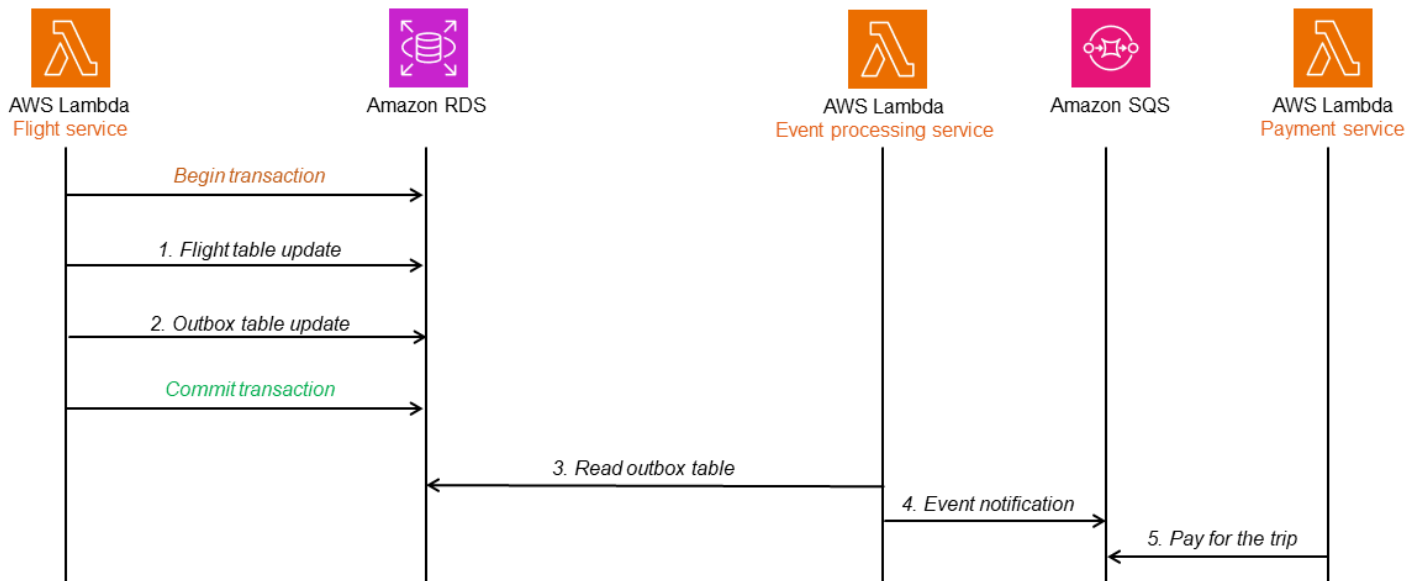
Para resolver esse problema, você pode usar uma tabela de caixa de saída ou captura de dados de alteração (CDC). As seções a seguir discutem essas duas opções e como você pode implementá-las usando os serviços da AWS.

Usar uma tabela de caixa de saída com um banco de dados relacional

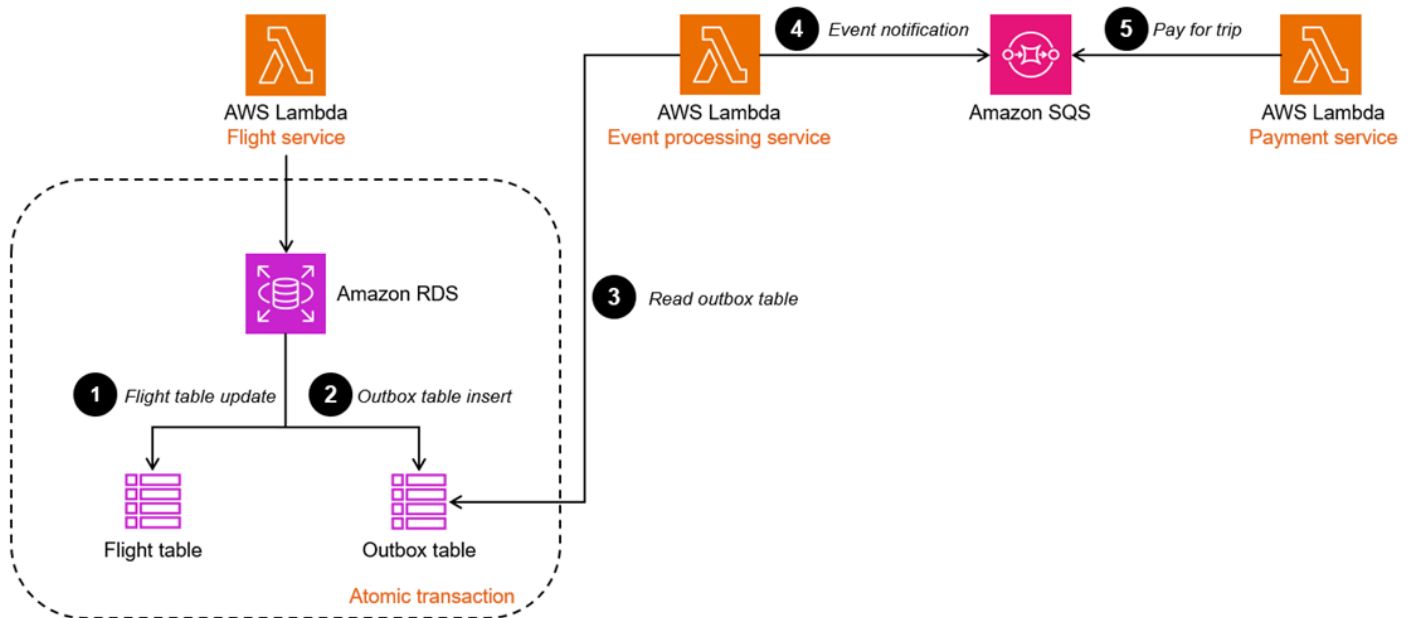
Uma tabela de caixa de saída armazena todos os eventos do serviço de voo com um registro de data e hora e um número de sequência.

Quando a tabela de voos é atualizada, a tabela da caixa de saída também é atualizada na mesma transação. Outro serviço (por ex., o serviço de processamento de eventos) lê a tabela da caixa de saída e envia o evento para o Amazon SQS. O Amazon SQS envia uma mensagem sobre o evento ao serviço de pagamento para processamento adicional. As [filas padrão do Amazon SQS](#) garantem que a mensagem seja entregue pelo menos uma vez e não se perca. No entanto, quando você usa as filas padrão do Amazon SQS, a mesma mensagem ou evento pode ser entregue mais de uma vez, portanto, você deve garantir que o serviço de notificação de eventos seja idempotente (ou seja, processar a mesma mensagem várias vezes não deve ter um efeito adverso). Se você precisar que a mensagem seja processada exatamente uma vez, com a ordenação de mensagens, você pode usar as filas [FIFO \(first in, first out\) do Amazon SQS](#).

Se houver falha na atualização da tabela de voo ou da tabela da caixa de saída, toda a transação será revertida, portanto, não haverá inconsistências de dados posteriores.



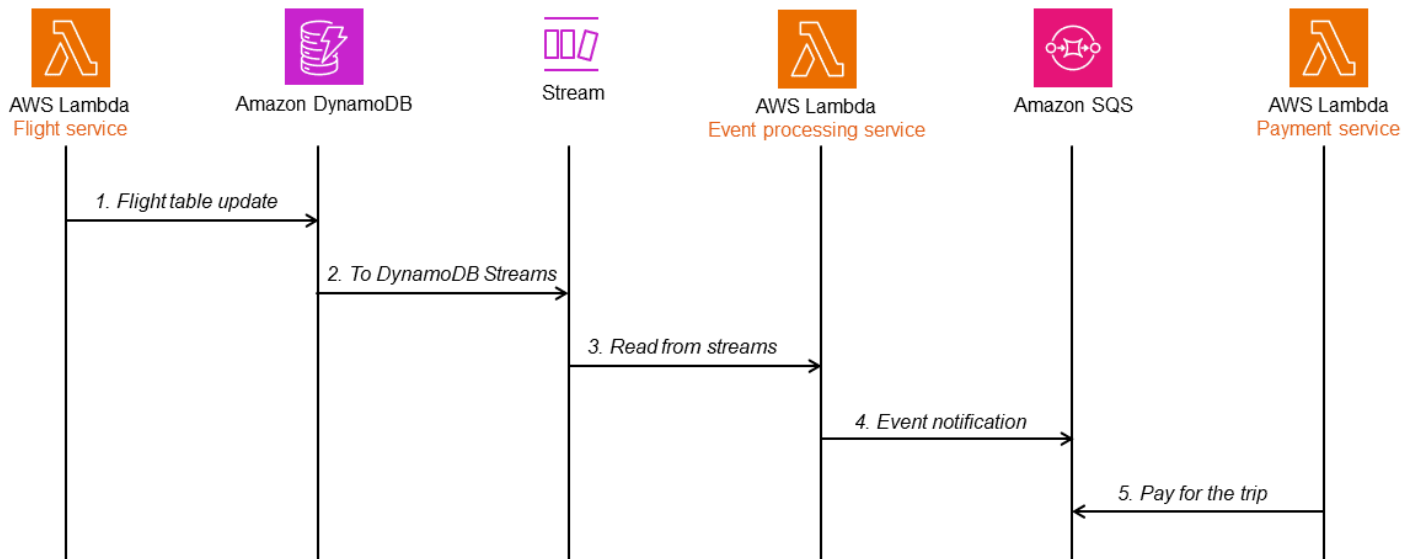
No diagrama a seguir, a arquitetura de caixa de saída transacional é implementada usando um banco de dados do Amazon RDS. Quando o serviço de processamento de eventos lê a tabela da caixa de saída, ele reconhece somente as linhas que fazem parte de uma transação confirmada (bem-sucedida) e, em seguida, coloca a mensagem do evento na fila do SQS, que é lida pelo serviço de pagamento para processamento adicional. Esse design resolve o problema das operações de gravação dupla e preserva a ordem das mensagens e dos eventos usando registros de data e hora e números de sequência.



Uso da captura de dados de alteração (CDC)

Alguns bancos de dados oferecem suporte à publicação de modificações em nível de item para capturar dados alterados. Você pode identificar os itens alterados e enviar uma notificação de evento adequadamente. Isso economiza a sobrecarga de criar outra tabela para rastrear as atualizações. O evento iniciado pelo serviço de voo é armazenado em outro atributo do mesmo item.

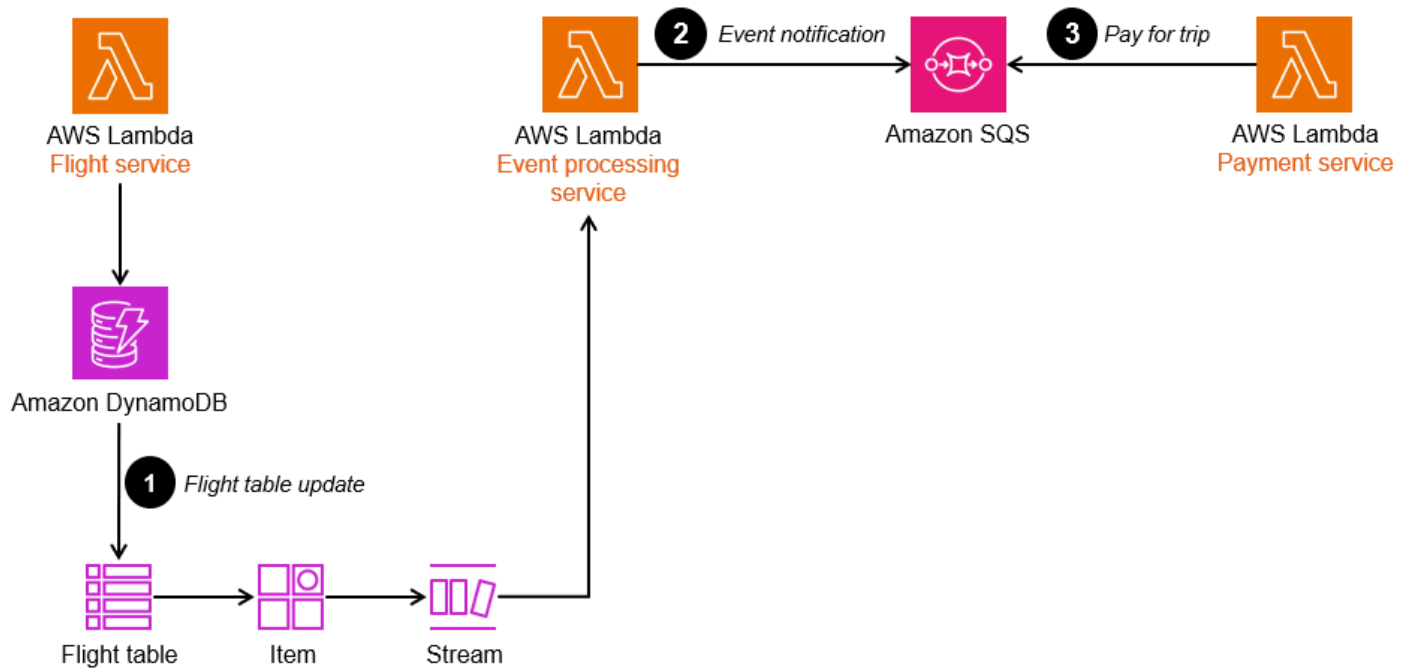
O [Amazon DynamoDB](#) é um banco de dados NoSQL de valor chave que oferece suporte às atualizações do CDC. No diagrama de sequência a seguir, o DynamoDB publica modificações em nível de item no Amazon DynamoDB Streams. O serviço de processamento de eventos lê os fluxos e publica a notificação do evento para o serviço de pagamento para processamento adicional.



O DynamoDB Streams captura o fluxo de informações relacionadas às mudanças no nível do item em uma tabela do DynamoDB usando uma sequência ordenada por tempo.

Você pode implementar um padrão de caixa de saída transacional ativando fluxos na tabela do DynamoDB. A função do Lambda para o serviço de processamento de eventos está associada a esses fluxos.

- Quando a tabela de voos é atualizada, os dados alterados são capturados pelo DynamoDB Streams, e o serviço de processamento de eventos pesquisa o fluxo em busca de novos registros.
- Quando novos registros de fluxo são disponibilizados, a função do Lambda coloca a mensagem do evento de forma síncrona na fila do SQS para processamento adicional. Você pode adicionar um atributo ao item do DynamoDB para capturar o registro de data e hora e o número de sequência conforme necessário para melhorar a robustez da implementação.



Código de exemplo

Uso de uma tabela de outbox

O código de exemplo a seguir mostra como você pode implementar o padrão outbox transacional usando uma tabela de outbox. Para ver o código completo, consulte o [GitHubrepositório](#) deste exemplo.

O trecho de código a seguir salva a entidade `Flight` e o evento `Flight` no banco de dados em suas respectivas tabelas em uma única transação.

```

@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;
}
  
```

Um serviço separado é responsável por escanear regularmente a tabela de caixa de saída em busca de novos eventos, enviá-los para o Amazon SQS e excluí-los da tabela se o Amazon SQS responder com sucesso. A taxa de pesquisa é configurável no arquivo `application.properties`.

```
@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);
    }
}
```

Uso da captura de dados de alteração (CDC)

O código de exemplo desta seção mostra como você pode implementar o padrão outbox transacional usando os recursos de captura de dados de alteração (CDC) do DynamoDB. Para ver o código completo, consulte o [GitHubrepositório](#) deste exemplo.

O trecho de código do AWS Cloud Development Kit (AWS CDK) a seguir cria uma tabela de voos do DynamoDB e um fluxo de dados do Amazon Kinesis (`cdcStream`) e configura a tabela de voos para enviar todas as atualizações para o stream.

```

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,
    }
});

```

O trecho de código e a configuração a seguir definem uma função do Spring Cloud Stream que coleta as atualizações no fluxo do Kinesis e encaminha esses eventos para uma fila do SQS para processamento adicional.

```

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 repositório

Para uma implementação completa da arquitetura de amostra desse padrão, consulte o GitHub repositório em <https://github.com/aws-samples/transactional-outbox-pattern>.

Recursos

Referências

- [AWS Centro de Arquitetura da](#)
- [AWS Centro do desenvolvedor da](#)
- [A Amazon Builders' Library](#)

Ferramentas

- [AWS Well-Architected Tool](#)
- [AWS App2Container](#)
- [AWS Microservice Extractor for .NET](#)

Metodologias

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

Histórico do documento

A tabela a seguir descreve alterações significativas feitas neste guia. Se desejar receber notificações sobre futuras atualizações, inscreva-se em um [feed RSS](#).

Alteração	Descrição	Data
Novos padrões	Foram adicionados dois novos padrões: arquitetura hexagonal e scatter-gather .	7 de maio de 2024
Novos exemplos de código	Foi adicionado o código de exemplo para o caso de uso de captura de dados de alteração (CDC) ao padrão de caixa de saída transacional.	23 de fevereiro de 2024
Novos exemplos de código	<ul style="list-style-type: none">• Atualizou o padrão da caixa de saída transacional com código de amostra.• Foi removida a seção sobre padrões de orquestração e coreografia, que foi substituída pela coreografia saga e pela orquestração de saga.	16 de novembro de 2023
Novos padrões	Foram adicionados três novos padrões: coreografia saga , publicar/assinar e fornecimento de eventos .	14 de novembro de 2023
Atualizar	Foi atualizada a seção de implementação do padrão strangler fig .	2 de outubro de 2023

Publicação inicial

Essa primeira versão inclui oito padrões de design: camada anticorrupção (ACL), roteamento de API, disjuntor, orquestração e coreografia, repetição com recuo, orquestração de saga, strangler fig e caixa de saída transacional.

28 de julho de 2023

AWS Glossário de orientação prescritiva

A seguir estão os termos comumente usados em estratégias, guias e padrões fornecidos pela Orientação AWS Prescritiva. Para sugerir entradas, use o link Fornecer feedback no final do glossário.

Números

7 Rs

Sete estratégias comuns de migração para mover aplicações para a nuvem. Essas estratégias baseiam-se nos 5 Rs identificados pela Gartner em 2011 e consistem em:

- Refatorar/rearquitetar: mova uma aplicação e modifique sua arquitetura aproveitando ao máximo os recursos nativos de nuvem para melhorar a agilidade, a performance e a escalabilidade. Isso normalmente envolve a portabilidade do sistema operacional e do banco de dados. Exemplo: migrar seu banco de dados Oracle on-premises para o Amazon Aurora Edição Compatível com PostgreSQL.
- Redefinir a plataforma (mover e redefinir [mover e redefinir (lift-and-reshape)]): mova uma aplicação para a nuvem e introduza algum nível de otimização a fim de aproveitar os recursos da nuvem. Exemplo: migrar seu banco de dados Oracle on-premises para o Amazon Relational Database Service (Amazon RDS) para Oracle na Nuvem AWS.
- Recomprar (drop and shop): mude para um produto diferente, normalmente migrando de uma licença tradicional para um modelo SaaS. Exemplo: migrar seu sistema de gerenciamento de relacionamento com o cliente (CRM) para o Salesforce.com.
- Redefinir a hospedagem (mover sem alterações [lift-and-shift])mover uma aplicação para a nuvem sem fazer nenhuma alteração a fim de aproveitar os recursos da nuvem. Exemplo: migrar seu banco de dados Oracle on-premises para o Oracle em uma instância do EC2 na Nuvem AWS.
- Realocar (mover o hipervisor sem alterações [hypervisor-level lift-and-shift]): mover a infraestrutura para a nuvem sem comprar novo hardware, reescrever aplicações ou modificar suas operações existentes. Você migra servidores de uma plataforma on-premises para um serviço de nuvem para a mesma plataforma. Exemplo: Migrar um Microsoft Hyper-V aplicativo para o. AWS
- Reter (revisitar): mantenha as aplicações em seu ambiente de origem. Isso pode incluir aplicações que exigem grande refatoração, e você deseja adiar esse trabalho para um

momento posterior, e aplicações antigas que você deseja manter porque não há justificativa comercial para migrá-las.

- Retirar: desative ou remova aplicações que não são mais necessárias em seu ambiente de origem.

A

ABAC

Consulte [controle de acesso baseado em atributo](#).

serviços abstraídos

Veja [serviços gerenciados](#).

ACID

Veja [atomicidade, consistência, isolamento, durabilidade](#).

migração ativa-ativa

Um método de migração de banco de dados no qual os bancos de dados de origem e de destino são mantidos em sincronia (por meio de uma ferramenta de replicação bidirecional ou operações de gravação dupla), e ambos os bancos de dados lidam com transações de aplicações conectadas durante a migração. Esse método oferece suporte à migração em lotes pequenos e controlados, em vez de exigir uma substituição única. É mais flexível, mas exige mais trabalho do que a [migração ativa-passiva](#).

migração ativa-passiva

Um método de migração de banco de dados em que os bancos de dados de origem e de destino são mantidos em sincronia, mas somente o banco de dados de origem manipula as transações das aplicações conectadas, enquanto os dados são replicados no banco de dados de destino. O banco de dados de destino não aceita nenhuma transação durante a migração.

AGGREGATE FUNCTION

Uma função SQL que opera em um grupo de linhas e calcula um único valor de retorno para o grupo. Exemplos de funções agregadas incluem SUM e MAX.

AI

Veja [inteligência artificial](#).

AIOps

Veja [operações de inteligência artificial](#).

anonimização

O processo de excluir permanentemente informações pessoais em um conjunto de dados. A anonimização pode ajudar a proteger a privacidade pessoal. Dados anônimos não são mais considerados dados pessoais.

antipadrões

Uma solução frequentemente usada para um problema recorrente em que a solução é contraproducente, ineficaz ou menos eficaz do que uma alternativa.

controle de aplicações

Uma abordagem de segurança que permite o uso somente de aplicações aprovadas para ajudar a proteger um sistema contra malware.

portfólio de aplicações

Uma coleção de informações detalhadas sobre cada aplicação usada por uma organização, incluindo o custo para criar e manter a aplicação e seu valor comercial. Essas informações são fundamentais para [o processo de descoberta e análise de portfólio](#) e ajudam a identificar e priorizar as aplicações a serem migradas, modernizadas e otimizadas.

inteligência artificial (IA)

O campo da ciência da computação que se dedica ao uso de tecnologias de computação para desempenhar funções cognitivas normalmente associadas aos humanos, como aprender, resolver problemas e reconhecer padrões. Para obter mais informações, consulte [O que é inteligência artificial?](#)

operações de inteligência artificial (AIOps)

O processo de usar técnicas de machine learning para resolver problemas operacionais, reduzir incidentes operacionais e intervenção humana e aumentar a qualidade do serviço. Para obter mais informações sobre como AIOps é usado na estratégia de AWS migração, consulte o [guia de integração de operações](#).

criptografia assimétrica

Um algoritmo de criptografia que usa um par de chaves, uma chave pública para criptografia e uma chave privada para descryptografia. É possível compartilhar a chave pública porque ela não é usada na descryptografia, mas o acesso à chave privada deve ser altamente restrito.

atomicidade, consistência, isolamento, durabilidade (ACID)

Um conjunto de propriedades de software que garantem a validade dos dados e a confiabilidade operacional de um banco de dados, mesmo no caso de erros, falhas de energia ou outros problemas.

controle de acesso por atributo (ABAC)

A prática de criar permissões minuciosas com base nos atributos do usuário, como departamento, cargo e nome da equipe. Para obter mais informações, consulte [ABAC AWS](#) na documentação AWS Identity and Access Management (IAM).

fonte de dados autorizada

Um local onde você armazena a versão principal dos dados, que é considerada a fonte de informações mais confiável. Você pode copiar dados da fonte de dados autorizada para outros locais com o objetivo de processar ou modificar os dados, como anonimizá-los, redigi-los ou pseudonimizá-los.

Zona de disponibilidade

Um local distinto dentro de um Região da AWS que está isolado de falhas em outras zonas de disponibilidade e fornece conectividade de rede barata e de baixa latência a outras zonas de disponibilidade na mesma região.

AWS Estrutura de adoção da nuvem (AWS CAF)

Uma estrutura de diretrizes e melhores práticas AWS para ajudar as organizações a desenvolver um plano eficiente e eficaz para migrar com sucesso para a nuvem. AWS O CAF organiza a orientação em seis áreas de foco chamadas perspectivas: negócios, pessoas, governança, plataforma, segurança e operações. As perspectivas de negócios, pessoas e governança têm como foco habilidades e processos de negócios; as perspectivas de plataforma, segurança e operações concentram-se em habilidades e processos técnicos. Por exemplo, a perspectiva das pessoas tem como alvo as partes interessadas que lidam com recursos humanos (RH), funções de pessoal e gerenciamento de pessoal. Nessa perspectiva, o AWS CAF fornece orientação para desenvolvimento, treinamento e comunicação de pessoas para ajudar a preparar a organização para a adoção bem-sucedida da nuvem. Para obter mais informações, consulte o [site da AWS CAF](#) e o [whitepaper da AWS CAF](#).

AWS Estrutura de qualificação da carga de trabalho (AWS WQF)

Uma ferramenta que avalia as cargas de trabalho de migração do banco de dados, recomenda estratégias de migração e fornece estimativas de trabalho. AWS O WQF está incluído com AWS

Schema Conversion Tool (AWS SCT). Ela analisa esquemas de banco de dados e objetos de código, código de aplicações, dependências e características de performance, além de fornecer relatórios de avaliação.

B

bot malicioso

Um [bot](#) destinado a causar disrupção ou danos a indivíduos ou organizações.

BCP

Veja [planejamento de continuidade de negócios](#)

gráfico de comportamento

Uma visualização unificada e interativa do comportamento e das interações de recursos ao longo do tempo. É possível usar um gráfico de comportamento com o Amazon Detective para examinar tentativas de login malsucedidas, chamadas de API suspeitas e ações similares. Para obter mais informações, consulte [Dados em um gráfico de comportamento](#) na documentação do Detective.

sistema big-endian

Um sistema que armazena o byte mais significativo antes. Veja também [endianness](#).

classificação binária

Um processo que prevê um resultado binário (uma de duas classes possíveis). Por exemplo, seu modelo de ML pode precisar prever problemas como “Este e-mail é ou não é spam?” ou “Este produto é um livro ou um carro?”

filtro de bloom

Uma estrutura de dados probabilística e eficiente em termos de memória que é usada para testar se um elemento é membro de um conjunto.

blue/green deployment (implantação azul/verde)

Uma estratégia de implantação em que você cria dois ambientes separados, mas idênticos. Você executa a versão atual da aplicação em um ambiente (azul) e a nova versão da aplicação no outro ambiente (verde). Essa estratégia ajuda você a reverter rapidamente com o mínimo de impacto.

bot

Uma aplicação de software que executa tarefas automatizadas na internet e simula a atividade ou interação humana. Alguns bots são úteis ou benéficos, como crawlers da web que indexam informações na internet. Outros bots, conhecidos como bots maliciosos, têm como objetivo causar interrupção ou danos a indivíduos ou organizações.

botnet

Redes de [bots](#) infectadas por [malware](#) e sob o controle de uma única parte, conhecidas como bot herder ou operador de bots. Os botnets são o mecanismo mais conhecido para escalar bots e seu impacto.

ramo

Uma área contida de um repositório de código. A primeira ramificação criada em um repositório é a ramificação principal. Você pode criar uma nova ramificação a partir de uma ramificação existente e, em seguida, desenvolver recursos ou corrigir bugs na nova ramificação. Uma ramificação que você cria para gerar um recurso é comumente chamada de ramificação de recurso. Quando o recurso estiver pronto para lançamento, você mesclará a ramificação do recurso de volta com a ramificação principal. Para obter mais informações, consulte [Sobre filiais](#) (GitHub documentação).

Acesso de emergência

Em circunstâncias excepcionais e por meio de um processo aprovado, um meio rápido para um usuário obter acesso a um Conta da AWS que ele normalmente não tem permissão para acessar. Para obter mais informações, consulte o indicador [Implement break-glass procedures](#) nas orientações do AWS Well-Architected.

estratégia brownfield

A infraestrutura existente em seu ambiente. Ao adotar uma estratégia brownfield para uma arquitetura de sistema, você desenvolve a arquitetura de acordo com as restrições dos sistemas e da infraestrutura atuais. Se estiver expandindo a infraestrutura existente, poderá combinar as estratégias brownfield e [greenfield](#).

cache do buffer

A área da memória em que os dados acessados com mais frequência são armazenados.

capacidade de negócios

O que uma empresa faz para gerar valor (por exemplo, vendas, atendimento ao cliente ou marketing). As arquiteturas de microsserviços e as decisões de desenvolvimento podem

ser orientadas por recursos de negócios. Para obter mais informações, consulte a seção [Organizados de acordo com as capacidades de negócios](#) do whitepaper [Executar microsserviços containerizados na AWS](#).

planejamento de continuidade de negócios (BCP)

Um plano que aborda o impacto potencial de um evento disruptivo, como uma migração em grande escala, nas operações e permite que uma empresa retome as operações rapidamente.

C

CAF

Veja [AWS Cloud Adoption Framework](#).

implantação canário

O lançamento lento e incremental de uma versão para usuários finais. Quando estiver confiante, você implanta a nova versão e substitui a versão atual por completo.

CCoE

Veja [Centro de Excelência da Nuvem](#).

CDC

Veja [captura de dados de alteração](#).

captura de dados de alterações (CDC)

O processo de rastrear alterações em uma fonte de dados, como uma tabela de banco de dados, e registrar metadados sobre a alteração. É possível usar o CDC para várias finalidades, como auditar ou replicar alterações em um sistema de destino para manter a sincronização.

engenharia do caos

Introduzir intencionalmente falhas ou eventos disruptivos para testar a resiliência de um sistema. Você pode usar [AWS Fault Injection Service \(AWS FIS\)](#) para realizar experimentos que estressam suas AWS cargas de trabalho e avaliar sua resposta.

CI/CD

Veja [integração e entrega contínuas](#).

classificação

Um processo de categorização que ajuda a gerar previsões. Os modelos de ML para problemas de classificação predizem um valor discreto. Os valores discretos são sempre diferentes uns dos outros. Por exemplo, um modelo pode precisar avaliar se há ou não um carro em uma imagem.

criptografia no lado do cliente

Criptografia de dados localmente, antes que o alvo os AWS service (Serviço da AWS) receba.

Centro de excelência em nuvem (CCoE)

Uma equipe multidisciplinar que impulsiona os esforços de adoção da nuvem em toda a organização, incluindo o desenvolvimento de práticas recomendadas de nuvem, a mobilização de recursos, o estabelecimento de cronogramas de migração e a liderança da organização em transformações em grande escala. Para obter mais informações, consulte as [publicações CCo E](#) no blog de estratégia Nuvem AWS corporativa.

computação em nuvem

A tecnologia de nuvem normalmente usada para armazenamento de dados remoto e gerenciamento de dispositivos de IoT. A computação em nuvem é normalmente conectada à tecnologia de [computação de borda](#).

modelo operacional em nuvem

Em uma organização de TI, o modelo operacional usado para criar, amadurecer e otimizar um ou mais ambientes de nuvem. Para obter mais informações, consulte [Criar seu modelo operacional de nuvem](#).

estágios de adoção da nuvem

As quatro fases pelas quais as organizações normalmente passam ao migrar para a Nuvem AWS:

- Projeto: executar alguns projetos relacionados à nuvem para fins de prova de conceito e aprendizado
- Fundação — Fazer investimentos fundamentais para escalar sua adoção da nuvem (por exemplo, criar uma landing zone, definir um CCo E, estabelecer um modelo de operações)
- Migração: migrar aplicações individuais
- Reinvenção: otimizar produtos e serviços e inovar na nuvem

Esses estágios foram definidos por Stephen Orban na postagem do blog [The Journey Toward Cloud-First & the Stages of Adoption](#) no blog de estratégia Nuvem AWS empresarial. Para obter

informações sobre como eles se relacionam com a estratégia de AWS migração, consulte o [guia de preparação para migração](#).

CMDB

Veja [banco de dados de gerenciamento de configuração](#).

repositório de código

Um local onde o código-fonte e outros ativos, como documentação, amostras e scripts, são armazenados e atualizados por meio de processos de controle de versão. Os repositórios de nuvem comuns incluem o GitHub ou o Bitbucket Cloud. Cada versão do código é chamada de ramificação. Em uma estrutura de microsserviços, cada repositório é dedicado a uma única peça de funcionalidade. Um único pipeline de CI/CD pode usar vários repositórios.

cache frio

Um cache de buffer que está vazio, não está bem preenchido ou contém dados obsoletos ou irrelevantes. Isso afeta a performance porque a instância do banco de dados deve ler da memória principal ou do disco, um processo que é mais lento do que a leitura do cache do buffer.

dados frios

Dados que raramente são acessados e geralmente são históricos. Ao consultar esse tipo de dados, consultas lentas geralmente são aceitáveis. Mover esses dados para níveis ou classes de armazenamento de baixo desempenho e menos caros pode reduzir os custos.

visão computacional (CV)

Um campo de [IA](#) que usa machine learning para analisar e extrair informações de formatos visuais, como vídeos e imagens digitais. Por exemplo, a Amazon SageMaker AI fornece algoritmos de processamento de imagem para CV.

desvio de configuração

Em uma workload, uma alteração de configuração em relação ao estado esperado. Isso pode fazer com que a workload se torne incompatível e, normalmente, é gradual e não intencional.

banco de dados de gerenciamento de configuração (CMDB)

Um repositório que armazena e gerencia informações sobre um banco de dados e seu ambiente de TI, incluindo componentes de hardware e software e suas configurações. Normalmente, os dados de um CMDB são usados no estágio de descoberta e análise do portfólio da migração.

pacote de conformidade

Um conjunto de AWS Config regras e ações de remediação que você pode montar para personalizar suas verificações de conformidade e segurança. Você pode implantar um pacote de conformidade como uma entidade única em uma Conta da AWS região ou em uma organização usando um modelo YAML. Para obter mais informações, consulte [Pacotes de conformidade na documentação](#). AWS Config

integração contínua e entrega contínua (CI/CD)

O processo de automatizar os estágios de origem, criação, teste, preparação e produção do processo de lançamento do software. CI/CD é comumente descrito como um pipeline. CI/CD pode ajudá-lo a automatizar processos, melhorar a produtividade, melhorar a qualidade do código e entregar com mais rapidez. Para obter mais informações, consulte [Benefícios da entrega contínua](#). CD também pode significar implantação contínua. Para obter mais informações, consulte [Entrega contínua versus implantação contínua](#).

CV

Veja [visão computacional](#).

D

dados em repouso

Dados estacionários em sua rede, por exemplo, dados que estão em um armazenamento.

classificação de dados

Um processo para identificar e categorizar os dados em sua rede com base em criticalidade e confidencialidade. É um componente crítico de qualquer estratégia de gerenciamento de riscos de segurança cibernética, pois ajuda a determinar os controles adequados de proteção e retenção para os dados. A classificação de dados é um componente do pilar de segurança no AWS Well-Architected Framework. Para obter mais informações, consulte [Classificação de dados](#).

desvio de dados

Uma variação significativa entre os dados de produção e os dados usados para treinar um modelo de ML ou uma alteração significativa nos dados de entrada ao longo do tempo. O desvio de dados pode reduzir a qualidade geral, a precisão e a imparcialidade das previsões do modelo de ML.

dados em trânsito

Dados que estão se movendo ativamente pela sua rede, como entre os recursos da rede.

data mesh

Um framework de arquitetura que fornece propriedade de dados distribuída e descentralizada com gerenciamento e governança centralizados.

minimização de dados

O princípio de coletar e processar apenas os dados estritamente necessários. Praticar a minimização de dados no Nuvem AWS pode reduzir os riscos de privacidade, os custos e a pegada de carbono de sua análise.

perímetro de dados

Um conjunto de proteções preventivas em seu AWS ambiente que ajudam a garantir que somente identidades confiáveis acessem recursos confiáveis das redes esperadas. Para obter mais informações, consulte [Construindo um perímetro de dados em AWS](#)

pré-processamento de dados

A transformação de dados brutos em um formato que seja facilmente analisado por seu modelo de ML. O pré-processamento de dados pode significar a remoção de determinadas colunas ou linhas e o tratamento de valores ausentes, inconsistentes ou duplicados.

proveniência dos dados

O processo de rastrear a origem e o histórico dos dados ao longo de seu ciclo de vida, por exemplo, como os dados foram gerados, transmitidos e armazenados.

titular dos dados

Um indivíduo cujos dados estão sendo coletados e processados.

data warehouse

Um sistema de gerenciamento de dados compatível com business intelligence, como analytics. Os data warehouses geralmente contêm grandes quantidades de dados históricos e geralmente são usados para consultas e análises.

linguagem de definição de dados (DDL)

Instruções ou comandos para criar ou modificar a estrutura de tabelas e objetos em um banco de dados.

linguagem de manipulação de dados (DML)

Instruções ou comandos para modificar (inserir, atualizar e excluir) informações em um banco de dados.

DDL

Veja [linguagem de definição de banco de dados](#).

deep ensemble

A combinação de vários modelos de aprendizado profundo para gerar previsões. Os deep ensembles podem ser usados para produzir uma previsão mais precisa ou para estimar a incerteza nas previsões.

Aprendizado profundo

Um subcampo do ML que usa várias camadas de redes neurais artificiais para identificar o mapeamento entre os dados de entrada e as variáveis-alvo de interesse.

defense-in-depth

Uma abordagem de segurança da informação na qual uma série de mecanismos e controles de segurança são cuidadosamente distribuídos por toda a rede de computadores para proteger a confidencialidade, a integridade e a disponibilidade da rede e dos dados nela contidos. Ao adotar essa estratégia AWS, você adiciona vários controles em diferentes camadas da AWS Organizations estrutura para ajudar a proteger os recursos. Por exemplo, uma defense-in-depth abordagem pode combinar autenticação multifatorial, segmentação de rede e criptografia.

administrador delegado

Em AWS Organizations, um serviço compatível pode registrar uma conta de AWS membro para administrar as contas da organização e gerenciar as permissões desse serviço. Essa conta é chamada de administrador delegado para esse serviço. Para obter mais informações e uma lista de serviços compatíveis, consulte [Serviços que funcionam com o AWS Organizations](#) na documentação do AWS Organizations .

implantação

O processo de criar uma aplicação, novos recursos ou correções de código disponíveis no ambiente de destino. A implantação envolve a implementação de mudanças em uma base de código e, em seguida, a criação e execução dessa base de código nos ambientes da aplicação

ambiente de desenvolvimento

Veja [ambiente](#).

controle detectivo

Um controle de segurança projetado para detectar, registrar e alertar após a ocorrência de um evento. Esses controles são uma segunda linha de defesa, alertando você sobre eventos de segurança que contornaram os controles preventivos em vigor. Para obter mais informações, consulte [Controles detectivos](#) em Como implementar controles de segurança na AWS.

mapeamento do fluxo de valor de desenvolvimento (DVSM)

Um processo usado para identificar e priorizar restrições que afetam negativamente a velocidade e a qualidade em um ciclo de vida de desenvolvimento de software. O DVSM estende o processo de mapeamento do fluxo de valor originalmente projetado para práticas de manufatura enxuta. Ele se concentra nas etapas e equipes necessárias para criar e movimentar valor por meio do processo de desenvolvimento de software.

gêmeo digital

Uma representação virtual de um sistema real, como um prédio, fábrica, equipamento industrial ou linha de produção. Os gêmeos digitais oferecem suporte à manutenção preditiva, ao monitoramento remoto e à otimização da produção.

tabela de dimensões

Em um [esquema em estrela](#), uma tabela menor que contém atributos de dados sobre dados quantitativos em uma tabela de fatos. Os atributos da tabela de dimensões geralmente são campos de texto ou números discretos que se comportam como texto. Esses atributos normalmente são usados para restringir consultas, filtrar e rotular conjuntos de resultados.

desastre

Um evento que impede que uma workload ou sistema cumpra seus objetivos de negócios em seu local principal de implantação. Esses eventos podem ser desastres naturais, falhas técnicas ou o resultado de ações humanas, como configuração incorreta não intencional ou ataque de malware.

Recuperação de desastres (RD)

A estratégia e o processo que você usa para minimizar o tempo de inatividade e a perda de dados causados por um [desastre](#). Para obter mais informações, consulte [Recuperação de desastres de cargas de trabalho em AWS: Recuperação na nuvem no AWS Well-Architected Framework](#).

DML

Veja [linguagem de manipulação de banco de dados](#).

design orientado por domínio

Uma abordagem ao desenvolvimento de um sistema de software complexo conectando seus componentes aos domínios em evolução, ou principais metas de negócios, atendidos por cada componente. Esse conceito foi introduzido por Eric Evans em seu livro, *Design orientado por domínio: lidando com a complexidade no coração do software* (Boston: Addison-Wesley Professional, 2003). Para obter informações sobre como usar o design orientado por domínio com o padrão strangler fig, consulte [Modernizar incrementalmente os serviços web herdados do Microsoft ASP.NET \(ASMX\) usando contêineres e o Amazon API Gateway](#).

DR

Veja [recuperação de desastres](#).

Deteção da oscilação

Rastreamento de desvios de uma configuração de linha de base. Por exemplo, você pode usar AWS CloudFormation para [detectar desvios nos recursos do sistema](#) ou AWS Control Tower para [detectar mudanças em seu landing zone](#) que possam afetar a conformidade com os requisitos de governança.

DVSM

Veja [mapeamento do fluxo de valor de desenvolvimento](#).

E

EDA

Veja [análise exploratória de dados](#).

EDI

Veja [intercâmbio eletrônico de dados](#).

computação de borda

A tecnologia que aumenta o poder computacional de dispositivos inteligentes nas bordas de uma rede de IoT. Quando comparada com a [computação em nuvem](#), a computação de borda pode reduzir a latência da comunicação e melhorar o tempo de resposta.

intercâmbio eletrônico de dados (EDI)

A troca automatizada de documentos comerciais entre organizações. Para obter mais informações, consulte [O que é EDI \(Intercâmbio eletrônico de dados\)?](#).

criptografia

Um processo de computação que transforma dados de texto simples, legíveis por humanos, em texto cifrado.

chave de criptografia

Uma sequência criptográfica de bits aleatórios que é gerada por um algoritmo de criptografia. As chaves podem variar em tamanho, e cada chave foi projetada para ser imprevisível e exclusiva.

endianismo

A ordem na qual os bytes são armazenados na memória do computador. Os sistemas big-endian armazenam o byte mais significativo antes. Os sistemas little-endian armazenam o byte menos significativo antes.

endpoint

Veja [endpoint de serviço](#).

serviço de endpoint

Um serviço que pode ser hospedado em uma nuvem privada virtual (VPC) para ser compartilhado com outros usuários. Você pode criar um serviço de endpoint com AWS PrivateLink e conceder permissões a outros diretores Contas da AWS ou a AWS Identity and Access Management (IAM). Essas contas ou entidades principais podem se conectar ao serviço de endpoint de maneira privada criando endpoints da VPC de interface. Para obter mais informações, consulte [Criar um serviço de endpoint](#) na documentação do Amazon Virtual Private Cloud (Amazon VPC).

planejamento de recursos empresariais (ERP)

Um sistema que automatiza e gerencia os principais processos de negócios (como contabilidade, [MES](#) e gerenciamento de projetos) para uma empresa.

criptografia envelopada

O processo de criptografar uma chave de criptografia com outra chave de criptografia. Para obter mais informações, consulte [Criptografia de envelope](#) na documentação AWS Key Management Service (AWS KMS).

ambiente

Uma instância de uma aplicação em execução. Estes são tipos comuns de ambientes na computação em nuvem:

- ambiente de desenvolvimento: uma instância de uma aplicação em execução que está disponível somente para a equipe principal responsável pela manutenção da aplicação. Ambientes de desenvolvimento são usados para testar mudanças antes de promovê-las para ambientes superiores. Esse tipo de ambiente às vezes é chamado de ambiente de teste.
- ambientes inferiores: todos os ambientes de desenvolvimento para uma aplicação, como aqueles usados para compilações e testes iniciais.
- ambiente de produção: uma instância de uma aplicação em execução que os usuários finais podem acessar. Em um CI/CD pipeline, o ambiente de produção é o último ambiente de implantação.
- ambientes superiores: todos os ambientes que podem ser acessados por usuários que não sejam a equipe principal de desenvolvimento. Isso pode incluir um ambiente de produção, ambientes de pré-produção e ambientes para testes de aceitação do usuário.

epic

Em metodologias ágeis, categorias funcionais que ajudam a organizar e priorizar seu trabalho. Os epics fornecem uma descrição de alto nível dos requisitos e das tarefas de implementação. Por exemplo, os épicos de segurança AWS da CAF incluem gerenciamento de identidade e acesso, controles de detetive, segurança de infraestrutura, proteção de dados e resposta a incidentes. Para obter mais informações sobre epics na estratégia de migração da AWS, consulte o [guia de implementação do programa](#).

ERP

Veja [planejamento de recursos empresariais](#).

análise exploratória de dados (EDA)

O processo de analisar um conjunto de dados para entender suas principais características. Você coleta ou agrega dados e, em seguida, realiza investigações iniciais para encontrar padrões, detectar anomalias e verificar suposições. O EDA é realizado por meio do cálculo de estatísticas resumidas e da criação de visualizações de dados.

F

tabela de fatos

A tabela central em um [esquema em estrela](#). Ela armazena dados quantitativos sobre as operações comerciais. Normalmente, uma tabela de fatos contém dois tipos de colunas: as que contêm medidas e as que contêm uma chave externa para uma tabela de dimensões.

Antecipar-se à falha

Uma filosofia que usa testes frequentes e incrementais para reduzir o ciclo de vida do desenvolvimento. É uma parte essencial de uma abordagem ágil.

delimitação de isolamento contra falhas

No Nuvem AWS, um limite, como uma zona de disponibilidade, Região da AWS um plano de controle ou um plano de dados, que limita o efeito de uma falha e ajuda a melhorar a resiliência das cargas de trabalho. Para obter mais informações, consulte [AWS Fault Isolation Boundaries](#).

ramificação de recursos

Veja [ramificação](#).

recursos

Os dados de entrada usados para fazer uma previsão. Por exemplo, em um contexto de manufatura, os recursos podem ser imagens capturadas periodicamente na linha de fabricação.

importância do recurso

O quanto um recurso é importante para as previsões de um modelo. Isso geralmente é expresso como uma pontuação numérica que pode ser calculada por meio de várias técnicas, como Shapley Additive Explanations (SHAP) e gradientes integrados. Para obter mais informações, consulte [Interpretabilidade do modelo de aprendizado de máquina com AWS](#).

transformação de recursos

O processo de otimizar dados para o processo de ML, incluindo enriquecer dados com fontes adicionais, escalar valores ou extrair vários conjuntos de informações de um único campo de dados. Isso permite que o modelo de ML se beneficie dos dados. Por exemplo, se a data “2021-05-27 00:15:37” for dividida em “2021”, “maio”, “quinta” e “15”, isso poderá ajudar o algoritmo de aprendizado a aprender padrões diferenciados associados a diferentes componentes de dados.

prompt few shot

Fornecer a um [LLM](#) um pequeno número de exemplos que demonstram a tarefa e o resultado desejado antes de solicitar que ele execute uma tarefa semelhante. Essa técnica é uma aplicação do aprendizado em contexto, em que os modelos aprendem com exemplos (shots) incorporados aos prompts. Prompts few-shot podem ser eficazes para tarefas que exigem formatação, raciocínio ou conhecimento de domínio específicos. Veja também [prompts zero-shot](#).

FGAC

Veja [controle de acesso refinado](#).

Controle de acesso refinado (FGAC)

O uso de várias condições para permitir ou negar uma solicitação de acesso.

migração flash-cut

Um método de migração de banco de dados que usa replicação contínua de dados via [captura de dados de alteração](#) para migrar os dados no menor tempo possível, em vez de usar uma abordagem em fases. O objetivo é reduzir ao mínimo o tempo de inatividade.

FM

Veja [modelo de base](#).

modelo de base (FM)

Uma grande rede neural de aprendizado profundo que vem treinando em grandes conjuntos de dados generalizados e não rotulados. FMs são capazes de realizar uma ampla variedade de tarefas gerais, como entender a linguagem, gerar texto e imagens e conversar em linguagem natural. Para obter mais informações, consulte [O que são modelos de base?](#).

G

IA generativa

Um subconjunto de modelos de [IA](#) que foram treinados em grandes quantidades de dados e que podem usar um simples prompt de texto para criar novos artefatos e conteúdo, como imagens, vídeos, texto e áudio. Para obter mais informações, consulte [O que é IA generativa?](#).

bloqueio geográfico

Veja [restrições geográficas](#).

restrições geográficas (bloqueio geográfico)

Na Amazon CloudFront, uma opção para impedir que usuários em países específicos acessem distribuições de conteúdo. É possível usar uma lista de permissões ou uma lista de bloqueios para especificar países aprovados e banidos. Para obter mais informações, consulte [Restringir a distribuição geográfica do seu conteúdo](#) na CloudFront documentação.

Fluxo de trabalho do GitFlow

Uma abordagem na qual ambientes inferiores e superiores usam ramificações diferentes em um repositório de código-fonte. O fluxo de trabalho do Gitflow é considerado legado, e o [fluxo de trabalho trunk-based](#) é a abordagem moderna e preferencial.

golden image

Um snapshot de um sistema ou software usado como modelo para implantar novas instâncias desse sistema ou software. Por exemplo, na manufatura, uma golden image pode ser usada para provisionar software em vários dispositivos e ajudar a melhorar a velocidade, a escalabilidade e a produtividade nas operações de fabricação de dispositivos.

estratégia greenfield

A ausência de infraestrutura existente em um novo ambiente. Ao adotar uma estratégia greenfield para uma arquitetura de sistema, é possível selecionar todas as novas tecnologias sem a restrição da compatibilidade com a infraestrutura existente, também conhecida como [brownfield](#). Se estiver expandindo a infraestrutura existente, poderá combinar as estratégias brownfield e greenfield.

barreira de proteção

Uma regra de alto nível que ajuda a governar recursos, políticas e conformidade em todas as unidades organizacionais (OUs). Barreiras de proteção preventivas impõem políticas para garantir o alinhamento a padrões de conformidade. Elas são implementadas usando políticas de controle de serviço e limites de permissões do IAM. Barreiras de proteção detectivas detectam violações de políticas e problemas de conformidade e geram alertas para remediação. Eles são implementados usando AWS Config, AWS Security Hub CSPM, Amazon GuardDuty AWS Trusted Advisor, Amazon Inspector e verificações personalizadas AWS Lambda .

H

HA

Veja [alta disponibilidade](#).

migração heterogênea de bancos de dados

Migrar seu banco de dados de origem para um banco de dados de destino que usa um mecanismo de banco de dados diferente (por exemplo, Oracle para Amazon Aurora). A migração heterogênea geralmente faz parte de um esforço de redefinição da arquitetura, e converter

o esquema pode ser uma tarefa complexa. [O AWS fornece o AWS SCT](#) para ajudar nas conversões de esquemas.

alta disponibilidade (HA)

A capacidade de uma workload operar continuamente, sem intervenção, em caso de desafios ou desastres. Os sistemas AH são projetados para realizar o failover automático, oferecer consistentemente desempenho de alta qualidade e lidar com diferentes cargas e falhas com impacto mínimo no desempenho.

modernização de historiador

Uma abordagem usada para modernizar e atualizar os sistemas de tecnologia operacional (OT) para melhor atender às necessidades do setor de manufatura. Um historiador é um tipo de banco de dados usado para coletar e armazenar dados de várias fontes em uma fábrica.

dados de hold-out

Uma parte dos dados históricos rotulados que são retidos de um conjunto de dados usado para treinar um modelo de [machine learning](#). Você pode usar dados de hold-out para avaliar a performance do modelo comparando as previsões do modelo com os dados de retenção.

migração homogênea de bancos de dados

Migrar seu banco de dados de origem para um banco de dados de destino que compartilha o mesmo mecanismo de banco de dados (por exemplo, Microsoft SQL Server para Amazon RDS para SQL Server). A migração homogênea geralmente faz parte de um esforço de redefinição da hospedagem ou da plataforma. É possível usar utilitários de banco de dados nativos para migrar o esquema.

dados quentes

Dados acessados com frequência, como dados em tempo real ou dados translacionais recentes. Esses dados normalmente exigem uma camada ou classe de armazenamento de alto desempenho para fornecer respostas rápidas às consultas.

hotfix

Uma correção urgente para um problema crítico em um ambiente de produção. Devido à sua urgência, um hotfix geralmente é feito fora do fluxo de trabalho normal de DevOps lançamento.

período de hipercuidados

Imediatamente após a substituição, o período em que uma equipe de migração gerencia e monitora as aplicações migradas na nuvem para resolver quaisquer problemas. Normalmente,

a duração desse período é de 1 a 4 dias. No final do período de hipercuidados, a equipe de migração normalmente transfere a responsabilidade pelas aplicações para a equipe de operações de nuvem.

eu

laC

Veja [infraestrutura como código](#).

Política baseada em identidade

Uma política anexada a um ou mais diretores do IAM que define suas permissões no Nuvem AWS ambiente.

aplicação ociosa

Uma aplicação que tem um uso médio de CPU e memória entre 5 e 20% em um período de 90 dias. Em um projeto de migração, é comum retirar essas aplicações ou retê-las on-premises.

IloT

Veja [Internet das Coisas Industrial](#).

infraestrutura imutável

Um modelo que implanta uma nova infraestrutura para workloads de produção em vez de atualizar, aplicar patches ou modificar a infraestrutura existente. Infraestruturas imutáveis são inerentemente mais consistentes, confiáveis e preditivas do que [infraestruturas mutáveis](#). Para obter mais informações, consulte a prática recomendada [Implantar usando infraestrutura imutável](#) no AWS Well-Architected Framework.

VPC de entrada (admissão)

Em uma arquitetura de AWS várias contas, uma VPC que aceita, inspeciona e roteia conexões de rede de fora de um aplicativo. A [Arquitetura de Referência de AWS Segurança](#) recomenda configurar sua conta de rede com entrada, saída e inspeção VPCs para proteger a interface bidirecional entre seu aplicativo e a Internet em geral.

migração incremental

Uma estratégia de substituição na qual você migra a aplicação em pequenas partes, em vez de realizar uma única substituição completa. Por exemplo, é possível mover inicialmente

apenas alguns microsserviços ou usuários para o novo sistema. Depois de verificar se tudo está funcionando corretamente, mova os microsserviços ou usuários adicionais de forma incremental até poder descomissionar seu sistema herdado. Essa estratégia reduz os riscos associados a migrações de grande porte.

Indústria 4.0

Um termo que foi introduzido por [Klaus Schwab](#) em 2016 para se referir à modernização dos processos de manufatura por meio de avanços em conectividade, dados em tempo real, automação, analytics e IA/ML.

infraestrutura

Todos os recursos e ativos contidos no ambiente de uma aplicação.

Infraestrutura como código (IaC)

O processo de provisionamento e gerenciamento da infraestrutura de uma aplicação por meio de um conjunto de arquivos de configuração. A IaC foi projetada para ajudar você a centralizar o gerenciamento da infraestrutura, padronizar recursos e escalar rapidamente para que novos ambientes sejam reproduzíveis, confiáveis e consistentes.

Internet industrial das coisas (IIoT)

O uso de sensores e dispositivos conectados à Internet nos setores industriais, como manufatura, energia, automotivo, saúde, ciências biológicas e agricultura. Para obter mais informações, consulte [Criando uma estratégia de transformação digital industrial da Internet das Coisas \(IIoT\)](#).

VPC de inspeção

Em uma arquitetura de AWS várias contas, uma VPC centralizada que gerencia as inspeções do tráfego de rede entre VPCs (na mesma ou em diferentes Regiões da AWS) a Internet e as redes locais. A [Arquitetura de Referência de AWS Segurança](#) recomenda configurar sua conta de rede com entrada, saída e inspeção VPCs para proteger a interface bidirecional entre seu aplicativo e a Internet em geral.

Internet das coisas (IoT)

A rede de objetos físicos conectados com sensores ou processadores incorporados que se comunicam com outros dispositivos e sistemas pela Internet ou por uma rede de comunicação local. Para obter mais informações, consulte [O que é IoT?](#)

interpretabilidade

Uma característica de um modelo de machine learning que descreve o grau em que um ser humano pode entender como as previsões do modelo dependem de suas entradas. Para obter mais informações, consulte [Interpretabilidade do modelo de aprendizado de máquina com AWS](#).

IoT

Veja [Internet das Coisas](#).

Biblioteca de informações de TI (ITIL)

Um conjunto de práticas recomendadas para fornecer serviços de TI e alinhar esses serviços a requisitos de negócios. A ITIL fornece a base para o ITSM.

Gerenciamento de serviços de TI (ITSM)

Atividades associadas a design, implementação, gerenciamento e suporte de serviços de TI para uma organização. Para obter informações sobre a integração de operações em nuvem com ferramentas de ITSM, consulte o [guia de integração de operações](#).

ITIL

Veja [biblioteca de informações de TI](#).

ITSM

Veja [gerenciamento de serviços de TI](#).

L

controle de acesso baseado em etiqueta (LBAC)

Uma implementação do controle de acesso obrigatório (MAC) em que os usuários e os dados em si recebem explicitamente um valor de etiqueta de segurança. A interseção entre a etiqueta de segurança do usuário e a etiqueta de segurança dos dados determina quais linhas e colunas podem ser vistas pelo usuário.

zona de pouso

Uma landing zone é um AWS ambiente bem arquitetado, com várias contas, escalável e seguro. Um ponto a partir do qual suas organizações podem iniciar e implantar rapidamente workloads e aplicações com confiança em seu ambiente de segurança e infraestrutura. Para obter mais

informações sobre zonas de pouso, consulte [Configurar um ambiente da AWS com várias contas seguro e escalável](#).

grande modelo de linguagem (LLM)

Um modelo de [IA](#) de aprendizado profundo pré-treinado em uma grande quantidade de dados. Um LLM pode realizar várias tarefas, como responder a perguntas, resumir documentos, traduzir texto para outros idiomas e completar frases. Para obter mais informações, consulte [O que são LLMs](#).

migração de grande porte

Uma migração de 300 servidores ou mais.

LBAC

Veja [controle de acesso baseado em rótulo](#).

privilégio mínimo

A prática recomendada de segurança de conceder as permissões mínimas necessárias para executar uma tarefa. Para obter mais informações, consulte [Aplicar permissões de privilégios mínimos](#) na documentação do IAM.

mover sem alterações (lift-and-shift)

Veja [7 Rs](#).

sistema little-endian

Um sistema que armazena o byte menos significativo antes. Veja também [endianness](#).

LLM

Veja [grande modelo de linguagem](#).

ambientes inferiores

Veja [ambiente](#).

M

machine learning (ML)

Um tipo de inteligência artificial que usa algoritmos e técnicas para reconhecimento e aprendizado de padrões. O ML analisa e aprende com dados gravados, por exemplo, dados da

Internet das Coisas (IoT), para gerar um modelo estatístico baseado em padrões. Para obter mais informações, consulte [Machine learning](#).

ramificação principal

Veja [ramificação](#).

Malware

Software projetado para comprometer a segurança ou a privacidade do computador. O malware pode interromper os sistemas do computador, vaziar informações sensíveis ou obter acesso não autorizado. Exemplos de malware incluem vírus, worms, ransomware, cavalos de Troia, spyware e keyloggers.

Serviços gerenciados

Serviços da AWS para o qual AWS opera a camada de infraestrutura, o sistema operacional e as plataformas, e você acessa os endpoints para armazenar e recuperar dados. O Amazon Simple Storage Service (Amazon S3) e o Amazon DynamoDB são exemplos de serviços gerenciados. Eles também são conhecidos como serviços abstraídos.

sistema de execução de manufatura (MES)

Um sistema de software para rastrear, monitorar, documentar e controlar processos de produção que convertem matérias-primas em produtos acabados no chão de fábrica.

MAP

Veja [Programa de Aceleração da Migração](#).

mecanismo

Um processo completo em que você cria uma ferramenta, impulsiona a adoção da ferramenta e, em seguida, inspeciona os resultados para fazer ajustes. Um mecanismo é um ciclo que se reforça e se aprimora à medida que opera. Para obter mais informações, consulte [Construindo mecanismos](#) no AWS Well-Architected Framework.

conta de membro

Todos, Contas da AWS exceto a conta de gerenciamento, que fazem parte de uma organização em AWS Organizations. Uma conta só pode ser membro de uma organização de cada vez.

MES

Veja [sistema de execução de manufatura](#).

Transporte de Telemetria de Enfileiramento de Mensagens (MQTT)

[Um protocolo de comunicação leve machine-to-machine \(M2M\), baseado no padrão de publicação/assinatura, para dispositivos de IoT com recursos limitados.](#)

microsserviço

Um serviço pequeno e independente que se comunica de forma bem definida APIs e normalmente é de propriedade de equipes pequenas e independentes. Por exemplo, um sistema de seguradora pode incluir microsserviços que mapeiam as capacidades comerciais, como vendas ou marketing, ou subdomínios, como compras, reclamações ou análises. Os benefícios dos microsserviços incluem agilidade, escalabilidade flexível, fácil implantação, código reutilizável e resiliência. Para obter mais informações, consulte [Integração de microsserviços usando serviços sem AWS servidor.](#)

arquitetura de microsserviços

Uma abordagem à criação de aplicações com componentes independentes que executam cada processo de aplicação como um microsserviço. Esses microsserviços se comunicam por meio de uma interface bem definida usando leveza. APIs Cada microsserviço nessa arquitetura pode ser atualizado, implantado e escalado para atender à demanda por funções específicas de uma aplicação. Para obter mais informações, consulte [Implementação de microsserviços em. AWS](#)

Programa de Aceleração da Migração (MAP)

Um AWS programa que fornece suporte de consultoria, treinamento e serviços para ajudar as organizações a criar uma base operacional sólida para migrar para a nuvem e ajudar a compensar o custo inicial das migrações. O MAP inclui uma metodologia de migração para executar migrações legadas de forma metódica e um conjunto de ferramentas para automatizar e acelerar cenários comuns de migração.

migração em escala

O processo de mover a maior parte do portfólio de aplicações para a nuvem em ondas, com mais aplicações sendo movidas em um ritmo mais rápido a cada onda. Essa fase usa as práticas recomendadas e lições aprendidas nas fases anteriores para implementar uma fábrica de migração de equipes, ferramentas e processos para agilizar a migração de workloads por meio de automação e entrega ágeis. Esta é a terceira fase da [estratégia de migração para a AWS.](#)

fábrica de migração

Equipes multifuncionais que simplificam a migração de workloads por meio de abordagens automatizadas e ágeis. As equipes da fábrica de migração geralmente incluem operações,

analistas e proprietários de negócios, engenheiros de migração, desenvolvedores e DevOps profissionais que trabalham em sprints. Entre 20 e 50% de um portfólio de aplicações corporativas consiste em padrões repetidos que podem ser otimizados por meio de uma abordagem de fábrica. Para obter mais informações, consulte [discussão sobre fábricas de migração](#) e o [guia do Cloud Migration Factory](#) neste conjunto de conteúdo.

metadados de migração

As informações sobre a aplicação e o servidor necessárias para concluir a migração. Cada padrão de migração exige um conjunto de metadados de migração diferente. Exemplos de metadados de migração incluem a sub-rede, o grupo de segurança e AWS a conta de destino.

padrão de migração

Uma tarefa de migração repetível que detalha a estratégia de migração, o destino da migração e a aplicação ou o serviço de migração usado. Exemplo: rehoste a migração para o Amazon EC2 AWS com o Application Migration Service.

Avaliação de Portfólio para Migração (MPA)

Uma ferramenta on-line que fornece informações para validar o caso de negócios para migrar para a Nuvem AWS. O MPA fornece avaliação detalhada do portfólio (dimensionamento correto do servidor, preços, comparações de TCO, análise de custos de migração), bem como planejamento de migração (análise e coleta de dados de aplicações, agrupamento de aplicações, priorização de migração e planejamento de ondas). A [ferramenta MPA](#) (requer login) está disponível gratuitamente para todos os AWS consultores e consultores parceiros da APN.

Avaliação de Preparação para Migração (MRA)

O processo de obter insights sobre o status de prontidão de uma organização para a nuvem, identificar pontos fortes e fracos e criar um plano de ação para fechar as lacunas identificadas, usando o CAF. AWS Para mais informações, consulte o [guia de preparação para migração](#). A MRA é a primeira fase da [estratégia de migração para a AWS](#).

estratégia de migração

A abordagem usada para migrar uma workload para a Nuvem AWS. Para obter mais informações, veja a entrada [7 Rs](#) neste glossário e consulte [Mobilize sua organização para acelerar migrações em grande escala](#).

ML

Veja [machine learning](#).

modernização

Transformar uma aplicação desatualizada (herdada ou monolítica) e sua infraestrutura em um sistema ágil, elástico e altamente disponível na nuvem para reduzir custos, ganhar eficiência e aproveitar as inovações. Para obter mais informações, consulte [Strategy for modernizing applications in the Nuvem AWS](#).

avaliação de preparação para modernização

Uma avaliação que ajuda a determinar a preparação para modernização das aplicações de uma organização. Ela identifica benefícios, riscos e dependências e determina o quão bem a organização pode acomodar o estado futuro dessas aplicações. O resultado da avaliação é um esquema da arquitetura de destino, um roteiro que detalha as fases de desenvolvimento e os marcos do processo de modernização e um plano de ação para abordar as lacunas identificadas. Para obter mais informações, consulte [Evaluating modernization readiness for applications in the Nuvem AWS](#).

aplicações monolíticas (monólitos)

Aplicações que são executadas como um único serviço com processos fortemente acoplados. As aplicações monolíticas apresentam várias desvantagens. Se um recurso da aplicação apresentar um aumento na demanda, toda a arquitetura deverá ser escalada. Adicionar ou melhorar os recursos de uma aplicação monolítica também se torna mais complexo quando a base de código cresce. Para resolver esses problemas, é possível criar uma arquitetura de microsserviços. Para obter mais informações, consulte [Decompor monólitos em microsserviços](#).

MPA

Veja [Avaliação do Portfólio para Migração](#).

MQTT

Veja [Transporte de Telemetria de Enfileiramento de Mensagens](#).

classificação multiclasse

Um processo que ajuda a gerar previsões para várias classes (prevendo um ou mais de dois resultados). Por exemplo, um modelo de ML pode perguntar “Este produto é um livro, um carro ou um telefone?” ou “Qual categoria de produtos é mais interessante para este cliente?”

infraestrutura mutável

Um modelo que atualiza e modifica a infraestrutura existente para workloads de produção. Para melhorar a consistência, confiabilidade e previsibilidade, o AWS Well-Architected Framework recomenda o uso de infraestrutura [imutável](#) como uma prática recomendada.

O

OAC

Veja [controle de acesso de origem](#).

OAI

Veja [identidade de acesso de origem](#).

OCM

Veja [gerenciamento de alterações organizacionais](#).

migração offline

Um método de migração no qual a workload de origem é desativada durante o processo de migração. Esse método envolve tempo de inatividade prolongado e geralmente é usado para workloads pequenas e não críticas.

OI

Veja [integração de operações](#).

Ola

Veja [acordo de nível operacional](#).

migração online

Um método de migração no qual a workload de origem é copiada para o sistema de destino sem ser colocada offline. As aplicações conectadas à workload podem continuar funcionando durante a migração. Esse método envolve um tempo de inatividade nulo ou mínimo e normalmente é usado para workloads essenciais para a produção.

OPC-UA

Veja [Open Process Communications - Unified Architecture](#).

Open Process Communications - Unified Architecture (OPC-UA)

Um protocolo de comunicação machine-to-machine (M2M) para automação industrial. O OPC-UA fornece um padrão de interoperabilidade com esquemas de criptografia, autenticação e autorização de dados.

acordo de nível operacional (OLA)

Um acordo que esclarece o que os grupos funcionais de TI prometem oferecer uns aos outros para apoiar um acordo de serviço (SLA).

análise de prontidão operacional (ORR)

Uma lista de verificação de perguntas e práticas recomendadas associadas que ajudam você a entender, avaliar, prevenir ou reduzir o escopo de incidentes e possíveis falhas. Para obter mais informações, consulte [Operational Readiness Reviews \(ORR\)](#) no AWS Well-Architected Framework.

tecnologia operacional (TO)

Sistemas de hardware e software que trabalham com o ambiente físico para controlar operações, equipamentos e infraestrutura industriais. Na manufatura, a integração dos sistemas de tecnologia da informação (TI) e tecnologia operacional (TO) é o foco principal das transformações da [Indústria 4.0](#).

integração de operações (OI)

O processo de modernização das operações na nuvem, que envolve planejamento de preparação, automação e integração. Para obter mais informações, consulte o [guia de integração de operações](#).

trilha organizacional

Uma trilha criada por ela AWS CloudTrail registra todos os eventos de todas as Contas da AWS em uma organização em AWS Organizations. Essa trilha é criada em cada Conta da AWS que faz parte da organização e monitora a atividade em cada conta. Para obter mais informações, consulte [Criação de uma trilha para uma organização](#) na CloudTrail documentação.

gerenciamento de alterações organizacionais (OCM)

Uma estrutura para gerenciar grandes transformações de negócios disruptivas de uma perspectiva de pessoas, cultura e liderança. O OCM ajuda as organizações a se prepararem e fazerem a transição para novos sistemas e estratégias, acelerando a adoção de alterações, abordando questões de transição e promovendo mudanças culturais e organizacionais. Na estratégia de AWS migração, essa estrutura é chamada de aceleração de pessoas, devido à velocidade de mudança exigida nos projetos de adoção da nuvem. Para obter mais informações, consulte o [guia do OCM](#).

controle de acesso de origem (OAC)

Em CloudFront, uma opção aprimorada para restringir o acesso para proteger seu conteúdo do Amazon Simple Storage Service (Amazon S3). O OAC oferece suporte a todos os buckets S3 Regiões da AWS, criptografia do lado do servidor com AWS KMS (SSE-KMS) e solicitações dinâmicas ao bucket S3. PUT DELETE

Identidade do acesso de origem (OAI)

Em CloudFront, uma opção para restringir o acesso para proteger seu conteúdo do Amazon S3. Quando você usa o OAI, CloudFront cria um principal com o qual o Amazon S3 pode se autenticar. Os diretores autenticados podem acessar o conteúdo em um bucket do S3 somente por meio de uma distribuição específica. CloudFront Veja também [OAC](#), que fornece um controle de acesso mais granular e aprimorado.

ORR

Veja [análise de prontidão operacional](#).

OT

Veja [tecnologia operacional](#).

VPC de saída (egresso)

Em uma arquitetura de AWS várias contas, uma VPC que gerencia conexões de rede que são iniciadas de dentro de um aplicativo. A [Arquitetura de Referência de AWS Segurança](#) recomenda configurar sua conta de rede com entrada, saída e inspeção VPCs para proteger a interface bidirecional entre seu aplicativo e a Internet em geral.

P

limite de permissões

Uma política de gerenciamento do IAM anexada a entidades principais do IAM para definir as permissões máximas que o usuário ou perfil podem ter. Para obter mais informações, consulte [Limites de permissões](#) na documentação do IAM.

Informações de identificação pessoal (PII)

Informações que, quando visualizadas diretamente ou combinadas com outros dados relacionados, podem ser usadas para inferir razoavelmente a identidade de um indivíduo. Exemplos de PII incluem nomes, endereços e informações de contato.

PII

Veja [informações de identificação pessoal](#).

manual

Um conjunto de etapas predefinidas que capturam o trabalho associado às migrações, como a entrega das principais funções operacionais na nuvem. Um manual pode assumir a forma de scripts, runbooks automatizados ou um resumo dos processos ou etapas necessários para operar seu ambiente modernizado.

PLC

Veja [controlador lógico programável](#).

PLM

Veja [gerenciamento do ciclo de vida do produto](#).

política

Um objeto que pode definir permissões (veja [política baseada em identidade](#)), especificar condições de acesso (veja [política baseada em recurso](#)) ou definir as permissões máximas para todas as contas em uma organização no AWS Organizations (veja [política de controle de serviços](#)).

persistência poliglota

Escolher de forma independente a tecnologia de armazenamento de dados de um microsserviço com base em padrões de acesso a dados e outros requisitos. Se seus microsserviços tiverem a mesma tecnologia de armazenamento de dados, eles poderão enfrentar desafios de implementação ou apresentar baixa performance. Os microsserviços serão implementados com mais facilidade e alcançarão performance e escalabilidade melhores se usarem o armazenamento de dados mais bem adaptado às suas necessidades.

avaliação do portfólio

Um processo de descobrir, analisar e priorizar o portfólio de aplicações para planejar a migração. Para obter mais informações, consulte [Avaliar a preparação para a migração](#).

predicado

Uma condição de consulta que retorna `true` ou `false`, normalmente localizada em uma cláusula `WHERE`.

pushdown de predicados

Uma técnica de otimização de consultas de banco de dados que filtra os dados na consulta antes da transferência. Isso reduz a quantidade de dados que devem ser recuperados e processados do banco de dados relacional e melhora a performance das consultas.

controle preventivo

Um controle de segurança projetado para evitar que um evento ocorra. Esses controles são a primeira linha de defesa para ajudar a evitar acesso não autorizado ou alterações indesejadas em sua rede. Para obter mais informações, consulte [Controles preventivos](#) em Como implementar controles de segurança na AWS.

principal (entidade principal)

Uma entidade AWS que pode realizar ações e acessar recursos. Essa entidade geralmente é um usuário raiz para um Conta da AWS, uma função do IAM ou um usuário. Para obter mais informações, consulte Entidade principal em [Termos e conceitos de perfis](#) na documentação do IAM.

Privacidade por design

Uma abordagem em engenharia de sistemas que leva em consideração a privacidade em todo o processo de desenvolvimento.

zonas hospedadas privadas

Um contêiner que contém informações sobre como você deseja que o Amazon Route 53 responda às consultas de DNS para um domínio e seus subdomínios em um ou mais VPCs. Para obter mais informações, consulte [Como trabalhar com zonas hospedadas privadas](#) na documentação do Route 53.

controle proativo

Um [controle de segurança](#) desenvolvido para evitar a implantação de recursos não conformes. Esses controles verificam os recursos antes de serem provisionados. Se o recurso não estiver em conformidade com o controle, ele não será provisionado. Para obter mais informações, consulte o [guia de referência de controles](#) na AWS Control Tower documentação e consulte [Controles proativos](#) em Implementação de controles de segurança em AWS.

gerenciamento do ciclo de vida do produto (PLM)

O gerenciamento de dados e processos de um produto em todo o seu ciclo de vida, desde a concepção, o desenvolvimento e o lançamento, passando pelo crescimento e maturidade, até o declínio e a remoção.

ambiente de produção

Veja [ambiente](#).

controlador lógico programável (PLC)

Na manufatura, um computador altamente confiável e adaptável que monitora as máquinas e automatiza os processos de fabricação.

encadeamento de prompts

Uso da saída de um prompt do [LLM](#) como entrada para o próximo prompt para gerar respostas melhores. Essa técnica é usada para dividir uma tarefa complexa em subtarefas, ou para refinar ou expandir iterativamente uma resposta preliminar. Isso ajuda a melhorar a precisão e a relevância das respostas de um modelo e permite resultados mais granulares e personalizados.

pseudonimização

O processo de substituir identificadores pessoais em um conjunto de dados por valores de espaço reservado. A pseudonimização pode ajudar a proteger a privacidade pessoal. Os dados pseudonimizados ainda são considerados dados pessoais.

publish/subscribe (pub/sub)

Um padrão que permite comunicações assíncronas entre microsserviços para melhorar a escalabilidade e a capacidade de resposta. Por exemplo, em um [MES](#) baseado em microsserviços, um microsserviço pode publicar mensagens de eventos em um canal em que outros microsserviços possam assinar. O sistema pode adicionar novos microsserviços sem alterar o serviço de publicação.

Q

plano de consulta

Uma série de etapas, como instruções, usadas para acessar os dados em um sistema de banco de dados relacional SQL.

regressão de planos de consultas

Quando um otimizador de serviço de banco de dados escolhe um plano menos adequado do que escolhia antes de uma determinada alteração no ambiente de banco de dados ocorrer. Isso pode ser causado por alterações em estatísticas, restrições, configurações do ambiente, associações de parâmetros de consulta e atualizações do mecanismo de banco de dados.

R

Matriz RACI

Veja [responsável, aprovador, consultado, informado \(RACI\)](#).

RAG

Veja [geração aumentada via recuperação](#).

ransomware

Um software mal-intencionado desenvolvido para bloquear o acesso a um sistema ou dados de computador até que um pagamento seja feito.

Matriz RASCI

Veja [responsável, aprovador, consultado, informado \(RACI\)](#).

RCAC

Veja [controle de acesso por linha e coluna](#).

réplica de leitura

Uma cópia de um banco de dados usada somente para leitura. É possível encaminhar consultas para a réplica de leitura e reduzir a carga no banco de dados principal.

Redefinir arquitetura

Veja [7 Rs](#).

objetivo de ponto de recuperação (RPO).

O máximo período de tempo aceitável desde o último ponto de recuperação de dados.

Isso determina o que é considerado uma perda aceitável de dados entre o último ponto de recuperação e a interrupção do serviço.

objetivo de tempo de recuperação (RTO)

O máximo atraso aceitável entre a interrupção e a restauração do serviço.

refatorar

Veja [7 Rs](#).

Região

Uma coleção de AWS recursos em uma área geográfica. Cada um Região da AWS é isolado e independente dos outros para fornecer tolerância a falhas, estabilidade e resiliência. Para obter informações, consulte [Specify which Regiões da AWS your account can use](#).

regressão

Uma técnica de ML que prevê um valor numérico. Por exemplo, para resolver o problema de “Por qual preço esta casa será vendida?” um modelo de ML pode usar um modelo de regressão linear para prever o preço de venda de uma casa com base em fatos conhecidos sobre a casa (por exemplo, a metragem quadrada).

redefinir a hospedagem

Veja [7 Rs](#).

versão

Em um processo de implantação, o ato de promover mudanças em um ambiente de produção.

realocar

Veja [7 Rs](#).

redefinir a plataforma

Veja [7 Rs](#).

recomprar

Veja [7 Rs](#).

resiliência

A capacidade de uma aplicação de resistir ou se recuperar de interrupções. [Alta disponibilidade](#) e [recuperação de desastres](#) são considerações comuns ao planejar a resiliência na Nuvem AWS. Para obter mais informações, consulte [Nuvem AWS Resilience](#).

política baseada em recurso

Uma política associada a um recurso, como um bucket do Amazon S3, um endpoint ou uma chave de criptografia. Esse tipo de política especifica quais entidades principais têm acesso permitido, ações válidas e quaisquer outras condições que devem ser atendidas.

matriz responsável, accountable, consultada, informada (RACI)

Uma matriz que define as funções e responsabilidades de todas as partes envolvidas nas atividades de migração e nas operações de nuvem. O nome da matriz é derivado dos tipos de responsabilidade definidos na matriz: responsável (R), responsabilizável (A), consultado (C) e informado (I). O tipo de suporte (S) é opcional. Se você incluir suporte, a matriz será chamada de matriz RASCI e, se excluir, será chamada de matriz RACI.

controle responsivo

Um controle de segurança desenvolvido para conduzir a remediação de eventos adversos ou desvios em relação à linha de base de segurança. Para obter mais informações, consulte [Controles responsivos](#) em Como implementar controles de segurança na AWS.

reter

Veja [7 Rs](#).

Retirada

Veja [7 Rs](#).

Geração Aumentada de Recuperação (RAG)

Uma tecnologia de [IA generativa](#) em que um [LLM](#) faz referência a uma fonte de dados autorizada que está fora de suas fontes de dados de treinamento antes de gerar uma resposta. Por exemplo, um modelo RAG pode realizar uma pesquisa semântica na base de conhecimento ou nos dados personalizados de uma organização. Para obter mais informações, consulte [O que é RAG \(geração aumentada via recuperação\)?](#).

alternância

O processo de atualizar periodicamente um [segredo](#) para dificultar o acesso de um invasor às credenciais.

controle de acesso por linha e coluna (RCAC)

O uso de expressões SQL básicas e flexíveis que tenham regras de acesso definidas. O RCAC consiste em permissões de linha e máscaras de coluna.

RPO

Veja [objetivo de ponto de recuperação](#).

RTO

Veja [objetivo de tempo de recuperação](#).

runbook

Um conjunto de procedimentos manuais ou automatizados necessários para realizar uma tarefa específica. Eles são normalmente criados para agilizar operações ou procedimentos repetitivos com altas taxas de erro.

S

SAML 2.0

Um padrão aberto que muitos provedores de identidade (IdPs) usam. Esse recurso permite o login único federado (SSO), para que os usuários possam fazer login no Console de gerenciamento da AWS ou chamar as operações da AWS API sem que você precise criar um usuário no IAM para todos em sua organização. Para obter mais informações sobre a federação baseada em SAML 2.0, consulte [Sobre a federação baseada em SAML 2.0](#) na documentação do IAM.

SCADA

Veja [controle de supervisão e aquisição de dados](#).

SCP

Veja [política de controle de serviço](#).

secret

Em AWS Secrets Manager, informações confidenciais ou restritas, como uma senha ou credenciais de usuário, que você armazena de forma criptografada. Consiste no valor secreto e em seus metadados. O valor secreto pode ser binário, uma única string ou várias strings. Para obter mais informações, consulte [What's in a Secrets Manager secret?](#) na documentação do Secrets Manager.

segurança desde a concepção

Uma abordagem em engenharia de sistemas que leva em consideração a segurança em todo o processo de desenvolvimento.

controle de segurança

Uma barreira de proteção técnica ou administrativa que impede, detecta ou reduz a capacidade de uma ameaça explorar uma vulnerabilidade de segurança. Existem quatro tipos primários de controles de segurança: [preventivos](#), [detectivos](#), [responsivos](#) e [proativos](#).

hardening da segurança

O processo de reduzir a superfície de ataque para torná-la mais resistente a ataques. Isso pode incluir ações como remover recursos que não são mais necessários, implementar a prática recomendada de segurança de conceder privilégios mínimos ou desativar recursos desnecessários em arquivos de configuração.

sistema de gerenciamento de eventos e informações de segurança (SIEM)

Ferramentas e serviços que combinam sistemas de gerenciamento de informações de segurança (SIM) e gerenciamento de eventos de segurança (SEM). Um sistema SIEM coleta, monitora e analisa dados de servidores, redes, dispositivos e outras fontes para detectar ameaças e violações de segurança e gerar alertas.

automação de resposta de segurança

Uma ação predefinida e programada projetada para responder ou remediar automaticamente um evento de segurança. Essas automações servem como controles de segurança [responsivos](#) ou [detectivos](#) que ajudam você a implementar as melhores práticas AWS de segurança. Exemplos de ações de resposta automatizada incluem a modificação de um grupo de segurança da VPC, a aplicação de patches em uma instância do Amazon EC2 ou a alternância de credenciais.

Criptografia do lado do servidor

Criptografia dos dados em seu destino, por AWS service (Serviço da AWS) quem os recebe.

política de controle de serviços (SCP)

Uma política que fornece controle centralizado sobre as permissões de todas as contas em uma organização em AWS Organizations. SCPs defina barreiras ou estabeleça limites nas ações que um administrador pode delegar a usuários ou funções. Você pode usar SCPs como listas de permissão ou listas de negação para especificar quais serviços ou ações são permitidos ou proibidos. Para obter mais informações, consulte [Políticas de controle de serviço](#) na AWS Organizations documentação.

service endpoint (endpoint de serviço)

O URL do ponto de entrada para um AWS service (Serviço da AWS). Você pode usar o endpoint para se conectar programaticamente ao serviço de destino. Para obter mais informações, consulte [Endpoints do AWS service \(Serviço da AWS\)](#) na Referência geral da AWS.

acordo de serviço (SLA)

Um acordo que esclarece o que uma equipe de TI promete fornecer aos clientes, como tempo de atividade e performance do serviço.

indicador de nível de serviço (SLI)

Uma avaliação de um aspecto de performance de um serviço, como taxa de erro, disponibilidade ou throughput.

objetivo de nível de serviço (SLO)

Uma métrica alvo que representa a integridade de um serviço, conforme avaliado por um [indicador de nível de serviço](#).

modelo de responsabilidade compartilhada

Um modelo que descreve a responsabilidade com a qual você compartilha AWS pela segurança e conformidade na nuvem. AWS é responsável pela segurança da nuvem, enquanto você é responsável pela segurança na nuvem. Para obter mais informações, consulte o [Modelo de responsabilidade compartilhada](#).

SIEM

Veja [sistema de gerenciamento de eventos e informações de segurança](#).

ponto único de falha (SPOF)

Uma falha em um único componente crítico de uma aplicação que pode interromper o sistema.

SLA

Veja [acordo de serviço](#).

SLI

Veja [indicador de nível de serviço](#).

SLO

Veja [objetivo de nível de serviço](#).

split-and-seed modelo

Um padrão para escalar e acelerar projetos de modernização. À medida que novos recursos e lançamentos de produtos são definidos, a equipe principal se divide para criar novas equipes de produtos. Isso ajuda a escalar os recursos e os serviços da sua organização, melhora a produtividade do desenvolvedor e possibilita inovações rápidas. Para obter mais informações, consulte [Phased approach to modernizing applications in the Nuvem AWS](#).

SPOF

Veja [ponto único de falha](#).

esquema em estrela

Uma estrutura organizacional de banco de dados que usa uma grande tabela de fatos para armazenar dados transacionais ou medidos e usa uma ou mais tabelas dimensionais menores para armazenar atributos de dados. Essa estrutura foi projetada para ser usada em um [data warehouse](#) ou para fins de inteligência comercial.

padrão strangler fig

Uma abordagem à modernização de sistemas monolíticos que consiste em reescrever e substituir incrementalmente a funcionalidade do sistema até que o sistema herdado possa ser desativado. Esse padrão usa a analogia de uma videira que cresce e se torna uma árvore estabelecida e, eventualmente, supera e substitui sua hospedeira. O padrão foi [apresentado por Martin Fowler](#) como forma de gerenciar riscos ao reescrever sistemas monolíticos. Para ver um exemplo de como aplicar esse padrão, consulte [Modernizar incrementalmente os serviços Web herdados do Microsoft ASP.NET \(ASMX\) usando contêineres e o Amazon API Gateway](#).

sub-rede

Um intervalo de endereços IP na VPC. Cada sub-rede fica alocada em uma única zona de disponibilidade.

controle supervisão e aquisição de dados (SCADA)

Na manufatura, um sistema que usa hardware e software para monitorar ativos físicos e operações de produção.

symmetric encryption (criptografia simétrica)

Um algoritmo de criptografia que usa a mesma chave para criptografar e descriptografar dados.

testes sintéticos

Testar um sistema de forma que simule as interações do usuário para detectar possíveis problemas ou monitorar a performance. Você pode usar o [Amazon CloudWatch Synthetics](#) para criar esses testes.

prompt do sistema

Uma técnica para fornecer contexto, instruções ou orientações a um [LLM](#) a fim de direcionar seu comportamento. Os prompts do sistema ajudam a definir o contexto e a estabelecer regras para interações com os usuários.

T

tags

Pares de valores-chave que atuam como metadados para organizar seus recursos. AWS As tags podem ajudar você a gerenciar, identificar, organizar, pesquisar e filtrar recursos da . Para obter mais informações, consulte [Marcar seus recursos do AWS](#).

variável-alvo

O valor que você está tentando prever no ML supervisionado. Ela também é conhecida como variável de resultado. Por exemplo, em uma configuração de fabricação, a variável-alvo pode ser um defeito do produto.

lista de tarefas

Uma ferramenta usada para monitorar o progresso por meio de um runbook. Uma lista de tarefas contém uma visão geral do runbook e uma lista de tarefas gerais a serem concluídas. Para cada tarefa geral, ela inclui o tempo estimado necessário, o proprietário e o progresso.

ambiente de teste

Veja [ambiente](#).

treinamento

O processo de fornecer dados para que seu modelo de ML aprenda. Os dados de treinamento devem conter a resposta correta. O algoritmo de aprendizado descobre padrões nos dados de treinamento que mapeiam os atributos dos dados de entrada no destino (a resposta que você deseja prever). Ele gera um modelo de ML que captura esses padrões. Você pode usar o modelo de ML para obter previsões de novos dados cujo destino você não conhece.

gateway de trânsito

Um hub de trânsito de rede que você pode usar para interconectar sua rede com VPCs a rede local. Para obter mais informações, consulte [O que é um gateway de trânsito](#) na AWS Transit Gateway documentação.

fluxo de trabalho baseado em troncos

Uma abordagem na qual os desenvolvedores criam e testam recursos localmente em uma ramificação de recursos e, em seguida, mesclam essas alterações na ramificação principal. A ramificação principal é então criada para os ambientes de desenvolvimento, pré-produção e produção, sequencialmente.

Acesso confiável

Conceder permissões a um serviço que você especifica para realizar tarefas em sua organização AWS Organizations e em suas contas em seu nome. O serviço confiável cria um perfil vinculado ao serviço em cada conta, quando esse perfil é necessário, para realizar tarefas de gerenciamento para você. Para obter mais informações, consulte [Usando AWS Organizations com outros AWS serviços](#) na AWS Organizations documentação.

tuning (ajustar)

Alterar aspectos do processo de treinamento para melhorar a precisão do modelo de ML. Por exemplo, você pode treinar o modelo de ML gerando um conjunto de rótulos, adicionando rótulos e repetindo essas etapas várias vezes em configurações diferentes para otimizar o modelo.

equipe de duas pizzas

Uma pequena DevOps equipe que você pode alimentar com duas pizzas. Uma equipe de duas pizzas garante a melhor oportunidade possível de colaboração no desenvolvimento de software.

U

incerteza

Um conceito que se refere a informações imprecisas, incompletas ou desconhecidas que podem minar a confiabilidade dos modelos preditivos de ML. Há dois tipos de incertezas: a incerteza epistêmica é causada por dados limitados e incompletos, enquanto a incerteza aleatória é causada pelo ruído e pela aleatoriedade inerentes aos dados.

tarefas indiferenciadas

Também conhecido como trabalho pesado, trabalho necessário para criar e operar um aplicativo, mas que não fornece valor direto ao usuário final nem oferece vantagem competitiva. Exemplos de tarefas indiferenciadas incluem aquisição, manutenção e planejamento de capacidade.

ambientes superiores

Veja [ambiente](#).

V

aspiração

Uma operação de manutenção de banco de dados que envolve limpeza após atualizações incrementais para recuperar armazenamento e melhorar a performance.

controle de versões

Processos e ferramentas que rastreiam mudanças, como alterações no código-fonte em um repositório.

emparelhamento da VPC

Uma conexão entre duas VPCs que permite rotear o tráfego usando endereços IP privados. Para ter mais informações, consulte [O que é emparelhamento de VPC?](#) na documentação da Amazon VPC.

Vulnerabilidade

Uma falha de software ou hardware que compromete a segurança do sistema.

W

cache quente

Um cache de buffer que contém dados atuais e relevantes que são acessados com frequência. A instância do banco de dados pode ler do cache do buffer, o que é mais rápido do que ler da memória principal ou do disco.

dados mornos

Dados acessados raramente. Ao consultar esse tipo de dados, consultas moderadamente lentas geralmente são aceitáveis.

função de janela

Uma função SQL que executa um cálculo em um grupo de linhas que se relacionam de alguma forma com o registro atual. As funções de janela são úteis para processar tarefas, como calcular uma média móvel ou acessar o valor das linhas com base na posição relativa da linha atual.

workload

Uma coleção de códigos e recursos que geram valor empresarial, como uma aplicação voltada para o cliente ou um processo de backend.

workstreams

Grupos funcionais em um projeto de migração que são responsáveis por um conjunto específico de tarefas. Cada workstream é independente, mas oferece suporte aos outros workstreams do projeto. Por exemplo, o workstream de portfólio é responsável por priorizar aplicações, planejar ondas e coletar metadados de migração. O workstream de portfólio entrega esses ativos ao workstream de migração, que então migra os servidores e as aplicações.

WORM

Veja [gravação única e várias leituras](#).

WQF

Veja [AWS Workload Qualification Framework](#).

gravação única e várias leituras (WORM)

Um modelo de armazenamento que grava dados uma única vez e evita que os dados sejam excluídos ou modificados. Os usuários autorizados podem ler os dados quantas vezes forem necessárias, mas não podem alterá-los. Essa infraestrutura de armazenamento de dados é considerada [imutável](#).

Z

exploração de dia zero

Um ataque, normalmente malware, que tira proveito de uma [vulnerabilidade zero-day](#).

vulnerabilidade de dia zero

Uma falha ou vulnerabilidade não mitigada em um sistema de produção. Os agentes de ameaças podem usar esse tipo de vulnerabilidade para atacar o sistema. Os desenvolvedores frequentemente ficam cientes da vulnerabilidade como resultado do ataque.

prompt zero shot

Fornecer a um [LLM](#) instruções para realizar uma tarefa, mas sem exemplos (shots) que possam ajudar a orientá-lo. O LLM deve usar seu conhecimento pré-treinado para lidar com a tarefa. A

eficácia dos prompts zero-shot depende da complexidade da tarefa e da qualidade do prompt.

Veja também [prompts few-shot](#).

aplicação zumbi

Uma aplicação que tem um uso médio de CPU e memória inferior a 5%. Em um projeto de migração, é comum retirar essas aplicações.

As traduções são geradas por tradução automática. Em caso de conflito entre o conteúdo da tradução e da versão original em inglês, a versão em inglês prevalecerá.