Pola, arsitektur, dan implementasi desain cloud

AWS Bimbingan Preskriptif

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Bimbingan Preskriptif: Pola, arsitektur, dan implementasi desain cloud

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Merek dagang dan tampilan dagang Amazon tidak boleh digunakan sehubungan dengan produk atau layanan apa pun yang bukan milik Amazon, dengan cara apa pun yang dapat menyebabkan kebingungan di antara pelanggan, atau dengan cara apa pun yang merendahkan atau mendiskreditkan Amazon. Semua merek dagang lain yang tidak dimiliki oleh Amazon merupakan hak milik masing-masing pemiliknya, yang mungkin atau tidak terafiliasi, terkait dengan, atau disponsori oleh Amazon.

Table of Contents

Pola, arsitektur, dan implementasi desain cloud

Anitha Deenadayalan, Amazon Web Services ()AWS

Mei 2024 (riwayat dokumen)

Panduan ini memberikan panduan untuk menerapkan pola desain modernisasi yang umum digunakan dengan menggunakan AWS layanan. Semakin banyak aplikasi modern dirancang dengan menggunakan arsitektur layanan mikro untuk mencapai skalabilitas, meningkatkan kecepatan rilis, mengurangi ruang lingkup dampak perubahan, dan mengurangi regresi. Ini mengarah pada peningkatan produktivitas pengembang dan peningkatan kelincahan, inovasi yang lebih baik, dan peningkatan fokus pada kebutuhan bisnis. Arsitektur microservices juga mendukung penggunaan teknologi terbaik untuk layanan dan database, dan mempromosikan kode polyglot dan persistensi polyglot.

Secara tradisional, aplikasi monolitik berjalan dalam satu proses, menggunakan satu penyimpanan data, dan berjalan di server yang berskala vertikal. Sebagai perbandingan, aplikasi microservice modern berbutir halus, memiliki domain kesalahan independen, dijalankan sebagai layanan di seluruh jaringan, dan dapat menggunakan lebih dari satu penyimpanan data tergantung pada kasus penggunaan. Skala layanan secara horizontal, dan satu transaksi mungkin menjangkau beberapa database. Tim pengembangan harus fokus pada komunikasi jaringan, persistensi polyglot, penskalaan horizontal, konsistensi akhirnya, dan penanganan transaksi di seluruh penyimpanan data saat mengembangkan aplikasi dengan menggunakan arsitektur layanan mikro. Oleh karena itu, pola modernisasi sangat penting untuk memecahkan masalah yang umum terjadi dalam pengembangan aplikasi modern, dan mereka membantu mempercepat pengiriman perangkat lunak.

Panduan ini memberikan referensi teknis untuk arsitek cloud, prospek teknis, pemilik aplikasi dan bisnis, dan pengembang yang ingin memilih arsitektur cloud yang tepat untuk pola desain berdasarkan praktik terbaik yang dirancang dengan baik. Setiap pola yang dibahas dalam panduan ini membahas satu atau lebih skenario yang dikenal dalam arsitektur layanan mikro. Panduan ini membahas masalah dan pertimbangan yang terkait dengan setiap pola, menyediakan implementasi arsitektur tingkat tinggi, dan menjelaskan implementasi AWS untuk pola tersebut. GitHub Sampel sumber terbuka dan tautan lokakarya disediakan jika tersedia.

Panduan ini mencakup pola-pola berikut:

- Lapisan anti korupsi
- Pola perutean API:

  - Perutean nama host
  - Perutean jalur
  - Perutean header HTTP
- Pemutus sirkuit
- Sumber acara
- Arsitektur heksagonal
- Publikasi-berlangganan
- Coba lagi dengan backoff
- Pola saga:
  - Koreografi Saga
  - Orkestrasi Saga
- Scatter-kumpulkan
- Ara pencekik
- Kotak keluar transaksional

Hasil bisnis yang ditargetkan

Dengan menggunakan pola yang dibahas dalam panduan ini untuk memodernisasi aplikasi Anda, Anda dapat:

- Merancang dan menerapkan arsitektur yang andal, aman, efisien secara operasional yang dioptimalkan untuk biaya dan kinerja.
- Kurangi waktu siklus untuk kasus penggunaan yang memerlukan pola-pola ini, sehingga Anda dapat fokus pada tantangan khusus organisasi.
- Mempercepat pengembangan dengan menstandarisasi implementasi pola dengan menggunakan layanan AWS.
- Bantu pengembang Anda membangun aplikasi modern tanpa mewarisi utang teknis.

Pola lapisan anti-korupsi

Niat

Pola anti-korupsi (ACL) bertindak sebagai lapisan mediasi yang menerjemahkan semantik model domain dari satu sistem ke sistem lain. Ini menerjemahkan model konteks terbatas hulu (monolit) ke dalam model yang sesuai dengan konteks batas hilir (layanan mikro) sebelum mengkonsumsi kontrak komunikasi yang ditetapkan oleh tim hulu. Pola ini mungkin berlaku ketika konteks terbatas hilir berisi subdomain inti, atau model hulu adalah sistem warisan yang tidak dapat dimodifikasi. Ini juga mengurangi risiko transformasi dan gangguan bisnis dengan mencegah perubahan pada penelepon ketika panggilan mereka harus diarahkan secara transparan ke sistem target.

Motivasi

Selama proses migrasi, ketika aplikasi monolitik dimigrasikan ke layanan mikro, mungkin ada perubahan dalam semantik model domain dari layanan yang baru dimigrasi. Ketika fitur dalam monolit diperlukan untuk memanggil layanan mikro ini, panggilan harus diarahkan ke layanan yang dimigrasi tanpa memerlukan perubahan apa pun pada layanan panggilan. Pola ACL memungkinkan monolit untuk memanggil layanan mikro secara transparan dengan bertindak sebagai adaptor atau lapisan fasad yang menerjemahkan panggilan ke semantik yang lebih baru.

Penerapan

Pertimbangkan untuk menggunakan pola ini ketika:

- Aplikasi monolitik Anda yang ada harus berkomunikasi dengan fungsi yang telah dimigrasikan ke layanan mikro, dan model dan semantik domain layanan yang dimigrasi berbeda dari fitur aslinya.
- Dua sistem memiliki semantik yang berbeda dan perlu bertukar data, tetapi tidak praktis untuk memodifikasi satu sistem agar kompatibel dengan sistem lainnya.
- Anda ingin menggunakan pendekatan yang cepat dan disederhanakan untuk menyesuaikan satu sistem ke sistem lain dengan dampak minimal.
- Aplikasi Anda berkomunikasi dengan sistem eksternal.

Masalah dan pertimbangan

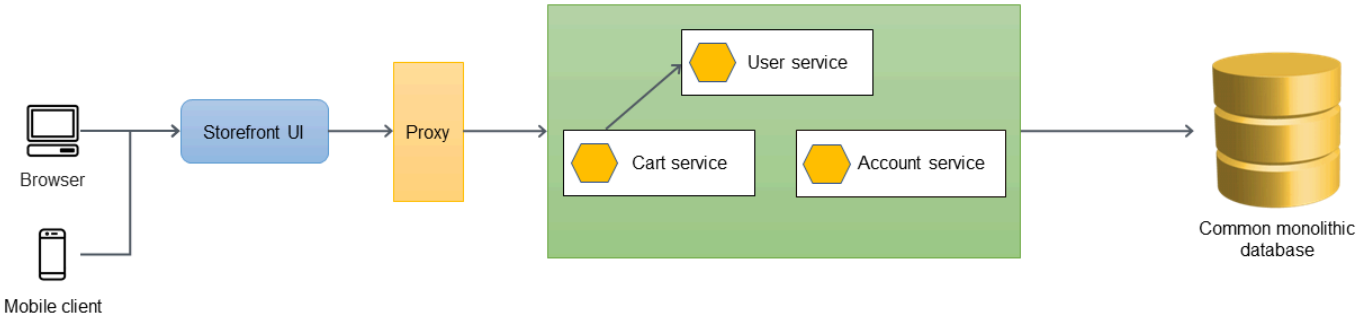
- Dependensi tim: Ketika layanan yang berbeda dalam suatu sistem dimiliki oleh tim yang berbeda, semantik model domain baru dalam layanan yang dimigrasi dapat menyebabkan perubahan dalam sistem panggilan. Namun, tim mungkin tidak dapat membuat perubahan ini secara terkoordinasi, karena mereka mungkin memiliki prioritas lain. ACL memisahkan panggilan dan menerjemahkan panggilan agar sesuai dengan semantik layanan baru, sehingga menghindari kebutuhan penelepon untuk membuat perubahan dalam sistem saat ini.
- Overhead operasional: Pola ACL membutuhkan upaya tambahan untuk mengoperasikan dan memelihara. Pekerjaan ini termasuk mengintegrasikan ACL dengan alat pemantauan dan peringatan, proses rilis, dan proses integrasi berkelanjutan dan pengiriman berkelanjutan (CI/CD).
- Titik kegagalan tunggal: Setiap kegagalan dalam ACL dapat membuat layanan target tidak dapat dijangkau, menyebabkan masalah aplikasi. Untuk mengurangi masalah ini, Anda harus membangun kemampuan coba lagi dan pemutus sirkuit. Lihat coba lagi dengan pola backoff dan pemutus sirkuit untuk memahami lebih lanjut tentang opsi ini. Menyiapkan peringatan dan pencatatan yang sesuai akan meningkatkan mean time to resolution (MTTR).
- Utang teknis: Sebagai bagian dari strategi migrasi atau modernisasi Anda, pertimbangkan apakah ACL akan menjadi solusi sementara atau sementara, atau solusi jangka panjang. Jika ini adalah solusi sementara, Anda harus mencatat ACL sebagai utang teknis dan menonaktifkannya setelah semua penelepon dependen telah dimigrasikan.
- Latensi: Lapisan tambahan dapat memperkenalkan latensi karena konversi permintaan dari satu antarmuka ke antarmuka lainnya. Kami menyarankan Anda menentukan dan menguji toleransi kinerja dalam aplikasi yang sensitif terhadap waktu respons sebelum Anda menerapkan ACL ke lingkungan produksi.
- Hambatan penskalaan: Dalam aplikasi dengan beban tinggi di mana layanan dapat menskalakan ke beban puncak, ACL dapat menjadi hambatan dan dapat menyebabkan masalah penskalaan. Jika skala layanan target sesuai permintaan, Anda harus merancang ACL agar sesuai dengan skala yang sesuai.
- Implementasi khusus layanan atau bersama: Anda dapat mendesain ACL sebagai objek bersama untuk mengonversi dan mengarahkan panggilan ke beberapa layanan atau kelas khusus layanan. Pertimbangkan latensi, penskalaan, dan toleransi kegagalan saat Anda menentukan jenis implementasi untuk ACL.

Implementasi

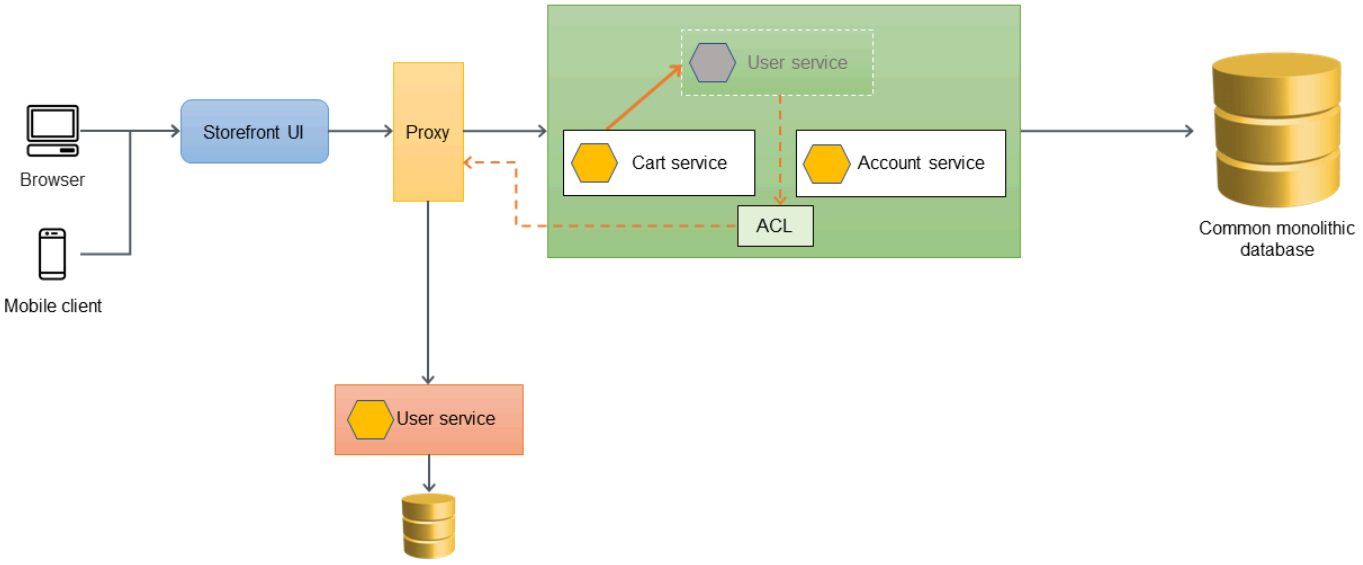
Anda dapat menerapkan ACL di dalam aplikasi monolitik Anda sebagai kelas yang khusus untuk layanan yang sedang dimigrasikan, atau sebagai layanan independen. ACL harus dinonaktifkan setelah semua layanan dependen telah dimigrasikan ke arsitektur layanan mikro.

Arsitektur tingkat tinggi

Dalam contoh arsitektur berikut, aplikasi monolitik memiliki tiga layanan: layanan pengguna, layanan keranjang, dan layanan akun. Layanan keranjang tergantung pada layanan pengguna, dan aplikasi menggunakan database relasional monolitik.

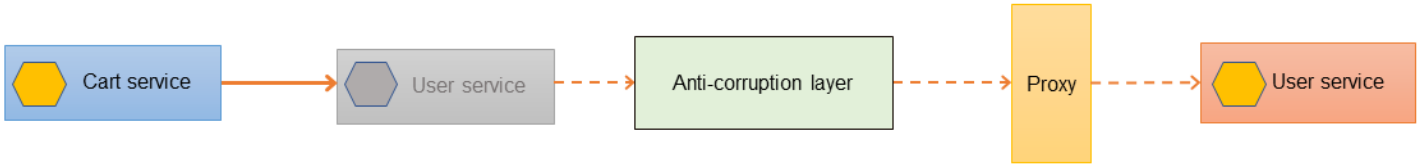


Dalam arsitektur berikut, layanan pengguna telah dimigrasikan ke layanan mikro baru. Layanan keranjang memanggil layanan pengguna, tetapi implementasinya tidak lagi tersedia dalam monolit. Kemungkinan juga antarmuka layanan yang baru dimigrasi tidak akan cocok dengan antarmuka sebelumnya, ketika berada di dalam aplikasi monolitik.



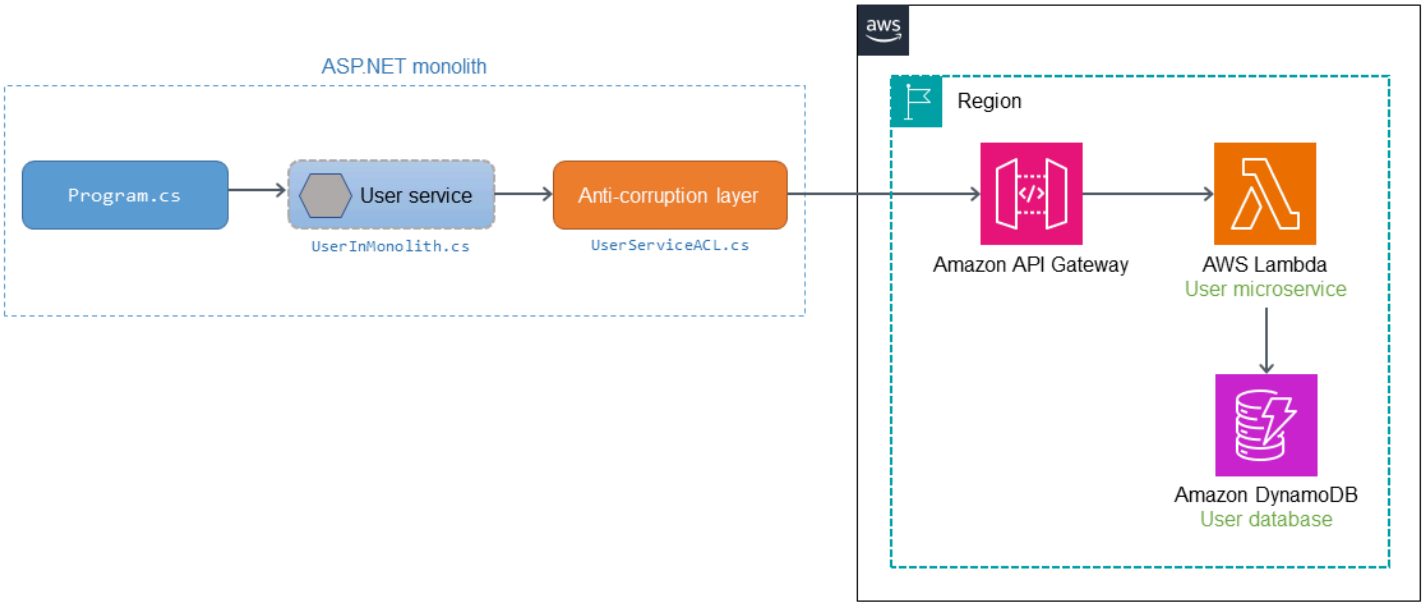
Jika layanan keranjang harus memanggil layanan pengguna yang baru dimigrasi secara langsung, ini akan memerlukan perubahan pada layanan keranjang dan pengujian menyeluruh dari aplikasi monolitik. Hal ini dapat meningkatkan risiko transformasi dan gangguan bisnis. Tujuannya adalah untuk meminimalkan perubahan pada fungsionalitas aplikasi monolitik yang ada.

Dalam hal ini, kami menyarankan Anda memperkenalkan ACL antara layanan pengguna lama dan layanan pengguna yang baru dimigrasi. ACL berfungsi sebagai adaptor atau fasad yang mengubah panggilan ke antarmuka yang lebih baru. ACL dapat diimplementasikan di dalam aplikasi monolitik sebagai kelas (misalnya, `UserServiceFacade` atau`UserServiceAdapter`) yang khusus untuk layanan yang dimigrasi. Lapisan anti-korupsi harus dinonaktifkan setelah semua layanan dependen telah dimigrasikan ke arsitektur layanan mikro.



Implementasi menggunakan AWS layanan

Diagram berikut menunjukkan bagaimana Anda dapat menerapkan contoh ACL ini dengan menggunakan AWS layanan.



Layanan mikro pengguna dimigrasikan keluar dari aplikasi monolitik ASP.NET dan digunakan sebagai fungsi di AWS. AWS Lambda Panggilan ke fungsi Lambda dirutekan melalui Amazon API

Gateway. ACL digunakan di monolit untuk menerjemahkan panggilan untuk beradaptasi dengan semantik layanan mikro pengguna.

Saat `Program.cs` memanggil layanan pengguna (`UserInMonolith.cs`) di dalam monolit, panggilan dirutekan ke ACL (). `UserServiceACL.cs` ACL menerjemahkan panggilan ke semantik dan antarmuka baru, dan memanggil layanan mikro melalui titik akhir API Gateway. Penelepon (`Program.cs`) tidak mengetahui terjemahan dan perutean yang terjadi di layanan pengguna dan ACL. Karena penelepon tidak menyadari perubahan kode, ada lebih sedikit gangguan bisnis dan pengurangan risiko transformasi.

Kode sampel

Cuplikan kode berikut memberikan perubahan pada layanan asli dan implementasi. `UserServiceACL.cs` Ketika permintaan diterima, layanan pengguna asli memanggil ACL. ACL mengonversi objek sumber agar sesuai dengan antarmuka layanan yang baru dimigrasi, memanggil layanan, dan mengembalikan respons ke pemanggil.

```
public class UserInMonolith: IUserInMonolith
{
    private readonly IACL _userServiceACL;
    public UserInMonolith(IACL userServiceACL) => (_userServiceACL) = (userServiceACL);
    public async Task<HttpStatusCode> UpdateAddress(UserDetails userDetails)
    {
        //Wrap the original object in the derived class
        var destUserDetails = new UserDetailsWrapped("user", userDetails);
        //Logic for updating address has been moved to a microservice
        return await _userServiceACL.CallMicroservice(destUserDetails);
    }
}

public class UserServiceACL: IACL
{
    static HttpClient _client = new HttpClient();
    private static string _apiGatewayDev = string.Empty;

    public UserServiceACL()
    {
        IConfiguration config = new
 ConfigurationBuilder().AddJsonFile(AppContext.BaseDirectory + "../../../
config.json").Build();
        _apiGatewayDev = config["APIGatewayURL:Dev"];
```

```
        _client.DefaultRequestHeaders.Accept.Add(new
 MediaTypeWithQualityHeaderValue("application/json"));
    }
    public async Task<HttpStatusCode> CallMicroservice(ISourceObject details)
    {
        _apiGatewayDev +=  "/" + details.ServiceName;
        Console.WriteLine(_apiGatewayDev);

        var userDetails = details as UserDetails;
        var userMicroserviceModel = new UserMicroserviceModel();
        userMicroserviceModel.UserId = userDetails.UserId;
        userMicroserviceModel.Address = userDetails.AddressLine1 + ", " +
 userDetails.AddressLine2;
        userMicroserviceModel.City = userDetails.City;
        userMicroserviceModel.State = userDetails.State;
        userMicroserviceModel.Country = userDetails.Country;

        if (Int32.TryParse(userDetails.ZipCode, out int zipCode))
        {
            userMicroserviceModel.ZipCode = zipCode;
            Console.WriteLine("Updated zip code");
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
            return HttpStatusCode.BadRequest;
        }

        var jsonString =
 JsonSerializer.Serialize<UserMicroserviceModel>(userMicroserviceModel);
        var payload = JsonSerializer.Serialize(userMicroserviceModel);
        var content = new StringContent(payload, Encoding.UTF8, "application/json");

        var response = await _client.PostAsync(_apiGatewayDev, content);
        return response.StatusCode;
    }
}
```

GitHub repositori

Untuk implementasi lengkap arsitektur sampel untuk pola ini, lihat GitHub repositori di. https://github.com/aws-samples/anti-corruption-layer-pattern

Konten terkait

- Pola ara pencekik
- Pola pemutus sirkuit
- Coba lagi dengan pola backoff

Pola perutean API

Dalam lingkungan pengembangan yang gesit, tim otonom (misalnya regu dan suku) memiliki satu atau lebih layanan yang mencakup banyak layanan mikro. Tim mengekspos layanan ini APIs untuk memungkinkan konsumen mereka berinteraksi dengan kelompok layanan dan tindakan mereka.

Ada tiga metode utama untuk mengekspos HTTP APIs ke konsumen hulu dengan menggunakan nama host dan jalur:

Metode	Deskripsi	Contoh
Perutean nama host	Paparkan setiap layanan sebagai nama host.	`billing.api.example.com`
Perutean jalur	Paparkan setiap layanan sebagai jalur.	`api.example.com/billing`
Perutean berbasis header	Paparkan setiap layanan sebagai header HTTP.	`x-example-action: something`

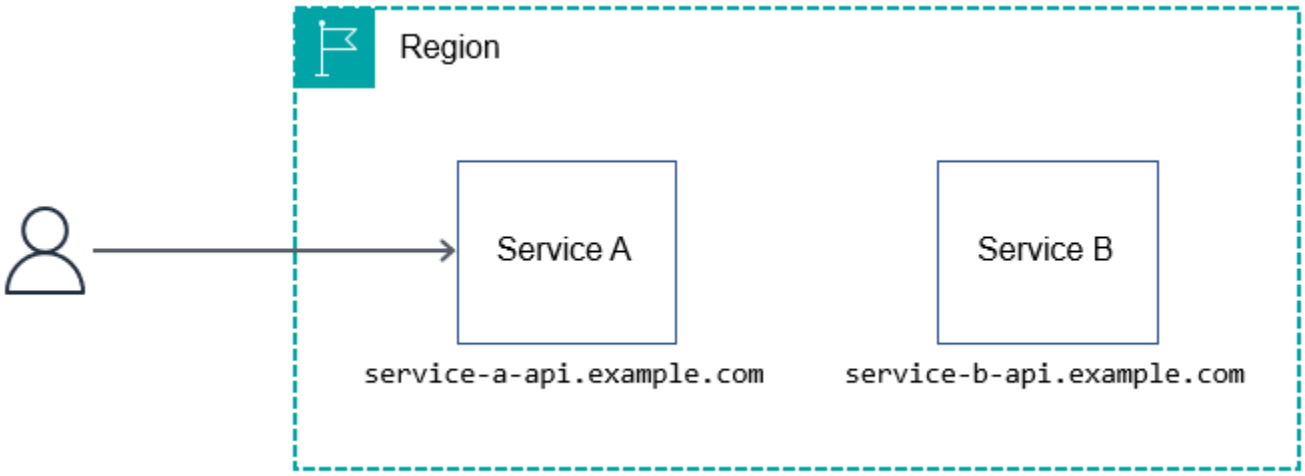
Bagian ini menguraikan kasus penggunaan umum untuk ketiga metode perutean ini dan trade-offnya untuk membantu Anda memutuskan metode mana yang paling sesuai dengan kebutuhan dan struktur organisasi Anda.

Pola perutean nama host

Routing dengan nama host adalah mekanisme untuk mengisolasi layanan API dengan memberikan masing-masing API nama hostnya sendiri; misalnya, atau. `service-a.api.example.com` `service-a.example.com`

Kasus penggunaan khas

Perutean dengan menggunakan nama host mengurangi jumlah gesekan dalam rilis, karena tidak ada yang dibagi antara tim layanan. Tim bertanggung jawab untuk mengelola segala sesuatu mulai dari entri DNS hingga operasi layanan dalam produksi.



Pro

Perutean nama host sejauh ini merupakan metode yang paling mudah dan terukur untuk perutean HTTP API. Anda dapat menggunakan AWS layanan apa pun yang relevan untuk membangun arsitektur yang mengikuti metode ini–Anda dapat membuat arsitektur dengan Amazon API Gateway AWS AppSync, Application Load Balancers dan Amazon Elastic Compute Cloud (EC2Amazon), atau layanan lain yang sesuai dengan HTTP.

Tim dapat menggunakan perutean nama host untuk sepenuhnya memiliki subdomain mereka. Ini juga membuatnya lebih mudah untuk mengisolasi, menguji, dan mengatur penerapan untuk spesifik Wilayah AWS atau versi; misalnya, atau. `region.service-a.api.example.com` `dev.region.service-a.api.example.com`

Kontra

Saat Anda menggunakan perutean nama host, konsumen Anda harus mengingat nama host yang berbeda untuk berinteraksi dengan setiap API yang Anda paparkan. Anda dapat mengurangi masalah ini dengan menyediakan SDK klien. Namun, klien SDKs datang dengan serangkaian tantangan mereka sendiri. Misalnya, mereka harus mendukung pembaruan bergulir, beberapa bahasa, pembuatan versi, mengkomunikasikan perubahan yang melanggar yang disebabkan oleh masalah keamanan atau perbaikan bug, dokumentasi, dan sebagainya.

Saat Anda menggunakan perutean nama host, Anda juga perlu mendaftarkan subdomain atau domain setiap kali Anda membuat layanan baru.

Pola perutean jalur

Perutean berdasarkan jalur adalah mekanisme pengelompokan beberapa atau semua APIs di bawah nama host yang sama, dan menggunakan URI permintaan untuk mengisolasi layanan; misalnya, atau. `api.example.com/service-a` `api.example.com/service-b`

Kasus penggunaan khas

Sebagian besar tim memilih metode ini karena mereka menginginkan arsitektur sederhana – pengembang harus mengingat hanya satu URL seperti `api.example.com` untuk berinteraksi dengan HTTP API. Dokumentasi API seringkali lebih mudah dicerna karena sering disimpan bersama alih-alih dibagi di berbagai portal atau. PDFs

Routing berbasis jalur dianggap sebagai mekanisme sederhana untuk berbagi API HTTP. Namun, ini melibatkan overhead operasional seperti konfigurasi, otorisasi, integrasi, dan latensi tambahan karena beberapa hop. Ini juga membutuhkan proses manajemen perubahan yang matang untuk memastikan bahwa kesalahan konfigurasi tidak mengganggu semua layanan.

Pada AWS, ada beberapa cara untuk berbagi API dan rute secara efektif ke layanan yang benar. Bagian berikut membahas tiga pendekatan: HTTP service reverse proxy, API Gateway, dan Amazon CloudFront. Tak satu pun dari pendekatan yang disarankan untuk menyatukan layanan API bergantung pada layanan hilir yang berjalan. AWS Layanan dapat berjalan di mana saja tanpa masalah atau pada teknologi apa pun, selama mereka kompatibel dengan HTTP.

Proxy terbalik layanan HTTP

Anda dapat menggunakan server HTTP seperti NGINX untuk membuat konfigurasi routing dinamis. Dalam arsitektur Kubernetes, Anda juga dapat membuat aturan ingress untuk mencocokkan jalur ke layanan. (Panduan ini tidak mencakup masuknya Kubernetes; lihat dokumentasi Kubernetes untuk informasi selengkapnya.)

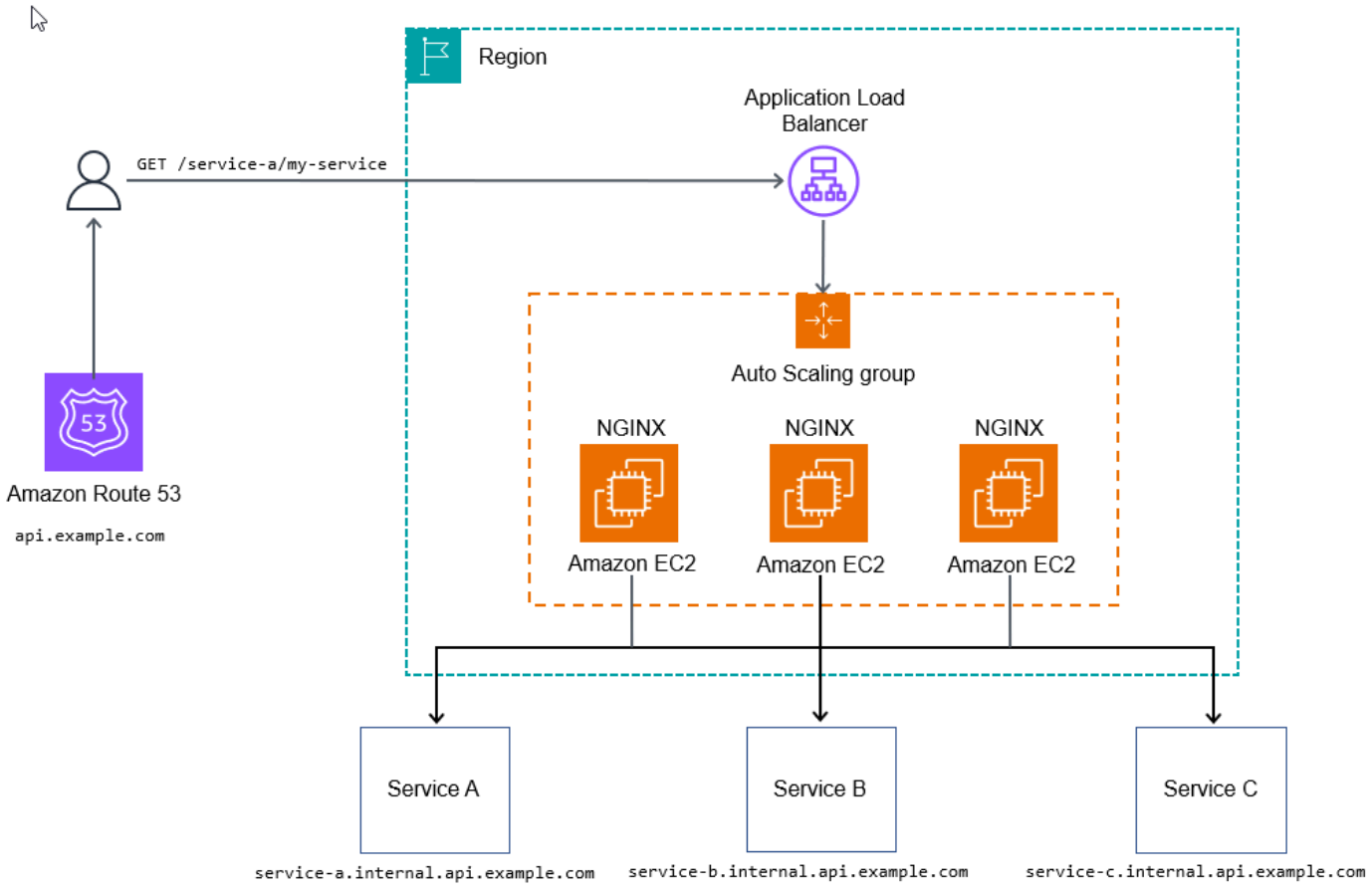
Konfigurasi berikut untuk NGINX secara dinamis memetakan permintaan HTTP ke. `api.example.com/my-service/` `my-service.internal.api.example.com`

```
server {
    listen  80;

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

```
    }
}
```

Diagram berikut menggambarkan layanan HTTP metode reverse proxy.



Pendekatan ini mungkin cukup untuk beberapa kasus penggunaan yang tidak menggunakan konfigurasi tambahan untuk mulai memproses permintaan, memungkinkan API hilir mengumpulkan metrik dan log.

Untuk bersiap-siap menghadapi kesiapan produksi operasional, Anda akan ingin dapat menambahkan observabilitas ke setiap tingkat tumpukan Anda, menambahkan konfigurasi tambahan, atau menambahkan skrip untuk menyesuaikan titik masuk API Anda untuk memungkinkan fitur yang lebih canggih seperti pembatasan tarif atau token penggunaan.

Pro

Tujuan utama dari metode proxy reverse layanan HTTP adalah untuk menciptakan pendekatan yang dapat diskalakan dan dikelola untuk menyatukan APIs ke dalam satu domain sehingga tampak

koheren untuk setiap konsumen API. Pendekatan ini juga memungkinkan tim layanan Anda untuk menerapkan dan mengelola sendiri APIs, dengan overhead minimal setelah penerapan. AWS layanan terkelola untuk penelusuran, seperti AWS X-Rayatau AWS WAF, masih berlaku di sini.

Kontra

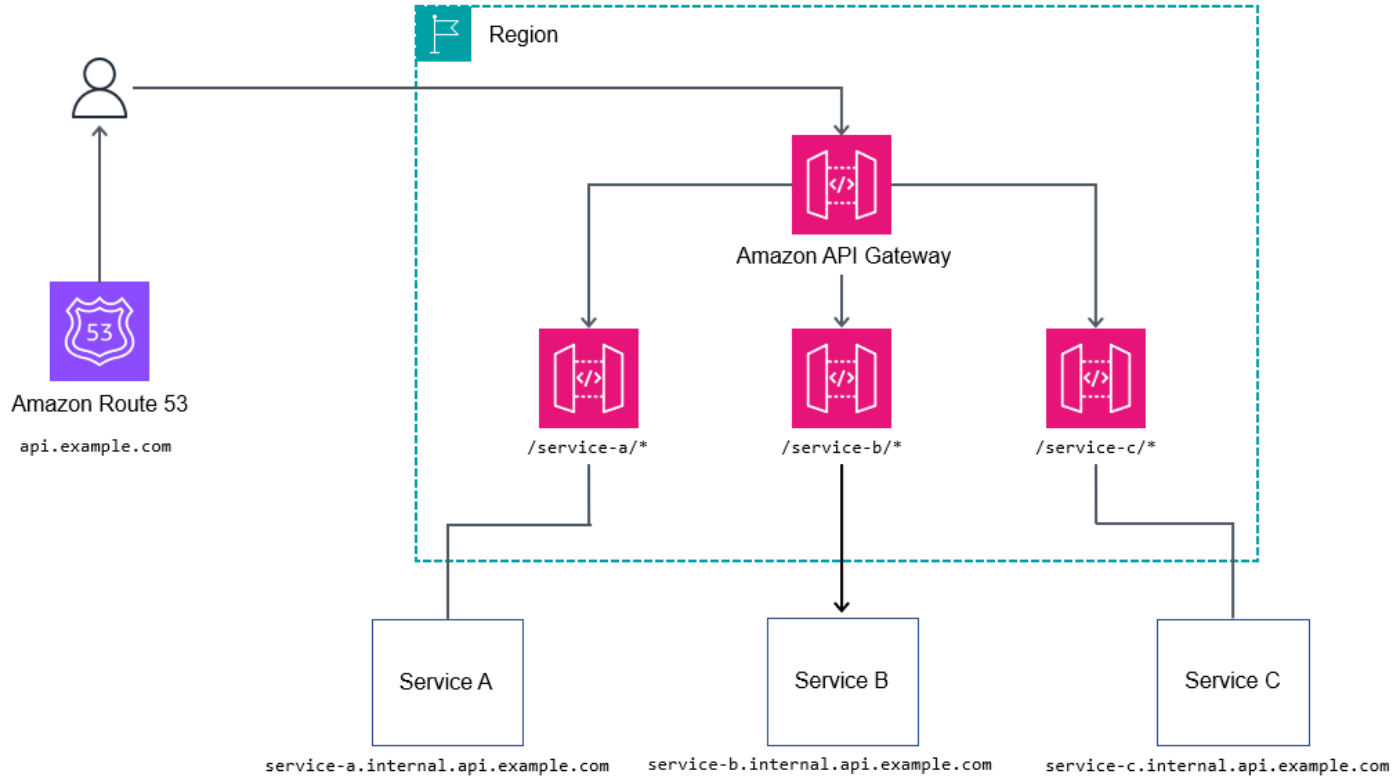
Kelemahan utama dari pendekatan ini adalah pengujian ekstensif dan manajemen komponen infrastruktur yang diperlukan, meskipun ini mungkin tidak menjadi masalah jika Anda memiliki tim rekayasa keandalan situs (SRE) di tempat.

Ada titik kritis biaya dengan metode ini. Pada volume rendah hingga menengah, ini lebih mahal daripada beberapa metode lain yang dibahas dalam panduan ini. Pada volume tinggi, ini sangat hemat biaya (sekitar 100 ribu transaksi per detik atau lebih baik).

API Gateway

Layanan Amazon API Gateway (REST APIs dan HTTP APIs) dapat merutekan lalu lintas dengan cara yang mirip dengan metode proxy terbalik layanan HTTP. Menggunakan gateway API dalam mode proxy HTTP menyediakan cara sederhana untuk membungkus banyak layanan ke titik masuk ke subdomain tingkat atas`api.example.com`, dan kemudian permintaan proxy ke layanan bersarang; misalnya,. `billing.internal.api.example.com`

Anda mungkin tidak ingin terlalu terperinci dengan memetakan setiap jalur di setiap layanan di gateway API root atau inti. Sebagai gantinya, pilih jalur wildcard seperti meneruskan permintaan `/billing/*` ke layanan penagihan. Dengan tidak memetakan setiap jalur di gateway API root atau inti, Anda mendapatkan lebih banyak fleksibilitas atas jalur Anda APIs, karena Anda tidak perlu memperbarui gateway API root dengan setiap perubahan API.



Pro

Untuk mengontrol alur kerja yang lebih kompleks, seperti mengubah atribut permintaan, REST APIs mengekspos Apache Velocity Template Language (VTL) untuk memungkinkan Anda memodifikasi permintaan dan respons. REST APIs dapat memberikan manfaat tambahan seperti ini:

- Auth N/Z dengan AWS Identity and Access Management (IAM), Amazon Cognito, atau otorisasi AWS Lambda
- AWS X-Ray untuk melacak
- Integrasi dengan AWS WAF
- Pembatasan tarif dasar
- Token penggunaan untuk mendorong konsumen ke berbagai tingkatan (lihat permintaan Throttle API untuk throughput yang lebih baik dalam dokumentasi API Gateway)

Kontra

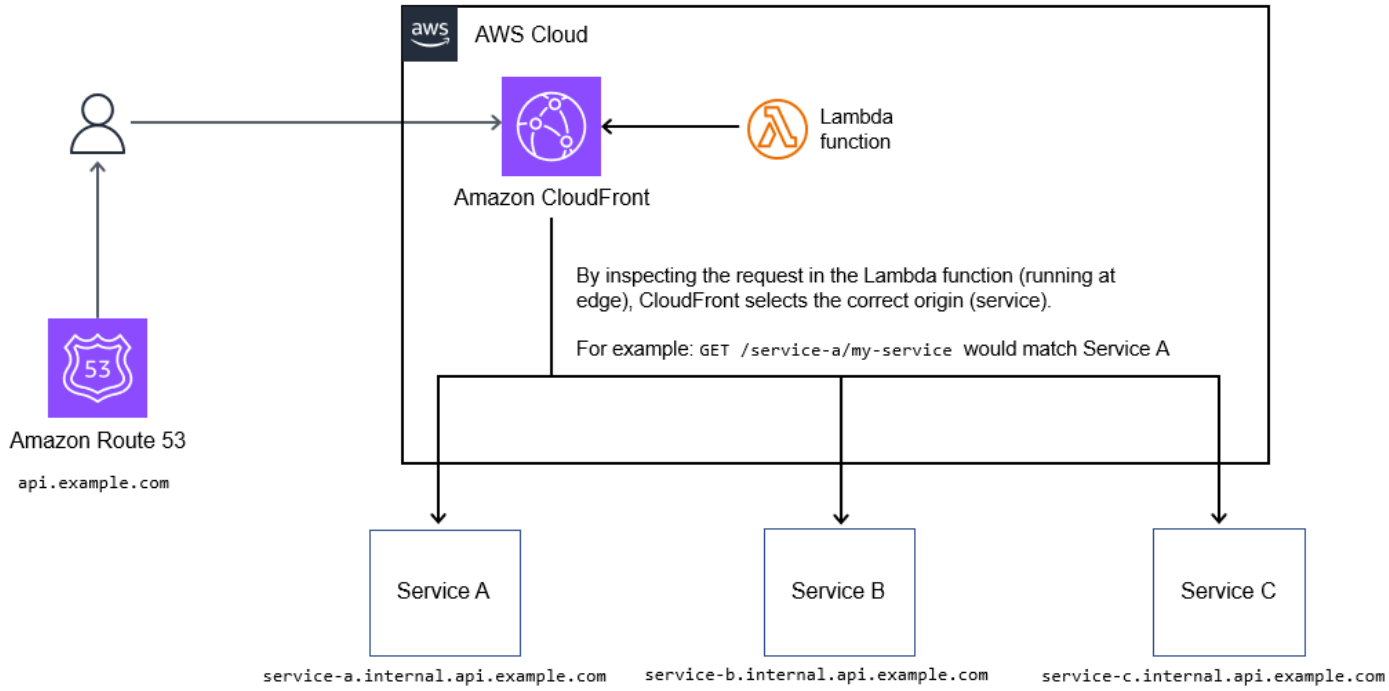
Pada volume tinggi, biaya mungkin menjadi masalah bagi beberapa pengguna.

CloudFront

Anda dapat menggunakan fitur pemilihan asal dinamis di Amazon CloudFront untuk memilih asal (layanan) secara kondisional untuk meneruskan permintaan. Anda dapat menggunakan fitur ini untuk merutekan sejumlah layanan melalui satu nama host seperti`api.example.com`.

Kasus penggunaan khas

Logika perutean hidup sebagai kode dalam fungsi Lambda @Edge, sehingga mendukung mekanisme perutean yang sangat dapat disesuaikan seperti pengujian A/B, rilis kenari, penandaan fitur, dan penulisan ulang jalur. Ini diilustrasikan dalam diagram berikut.



Pro

Jika Anda memerlukan respons API caching, metode ini adalah cara yang baik untuk menyatukan kumpulan layanan di balik satu titik akhir. Ini adalah metode hemat biaya untuk menyatukan koleksi. APIs

Juga, CloudFront mendukung enkripsi tingkat lapangan serta integrasi dengan AWS WAF untuk pembatasan tarif dasar dan dasar. ACLs

Kontra

Metode ini mendukung maksimal 250 asal (layanan) yang dapat disatukan. Batas ini cukup untuk sebagian besar penerapan, tetapi dapat menyebabkan masalah dengan sejumlah besar APIs saat Anda mengembangkan portofolio layanan Anda.

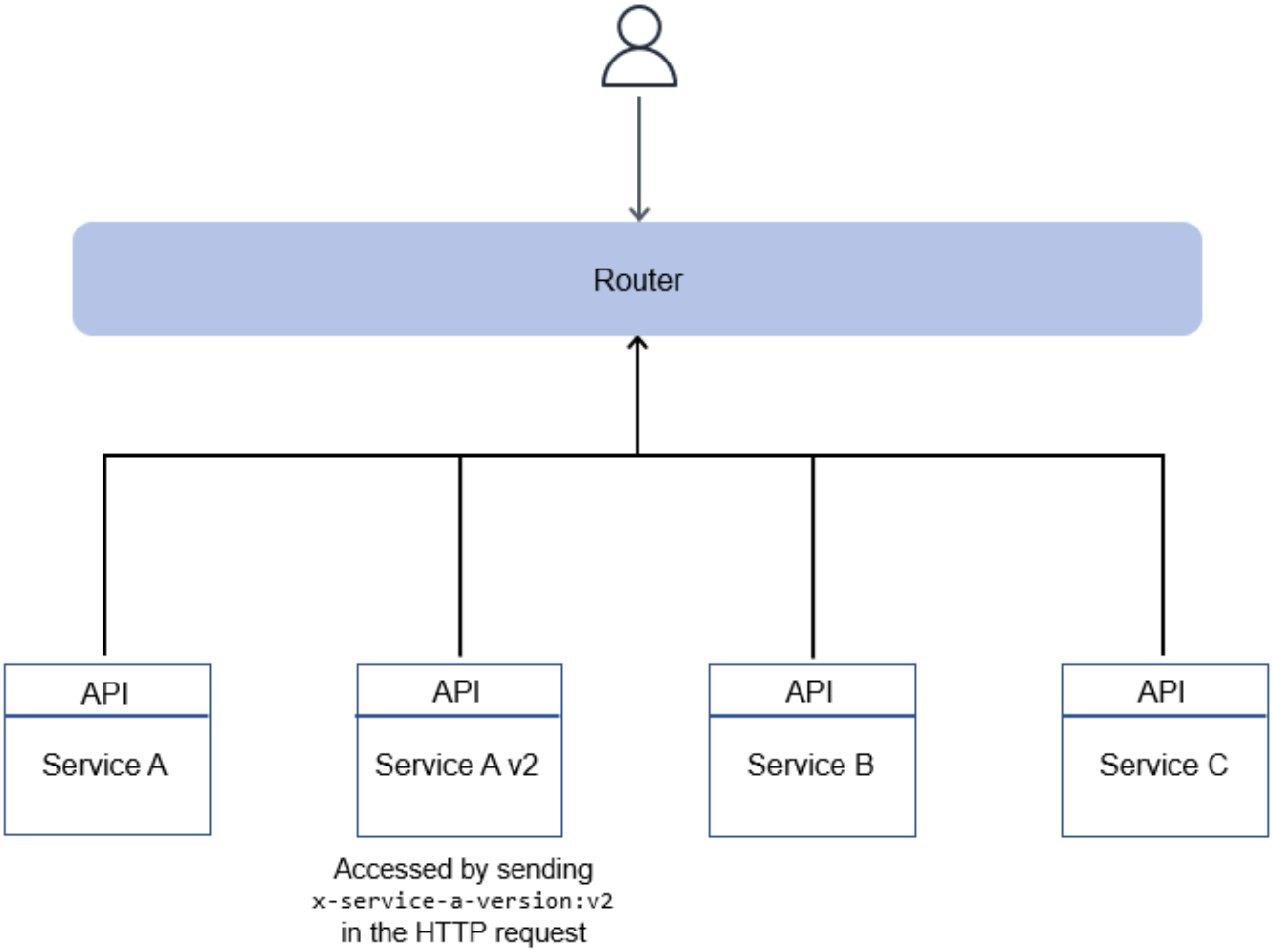
Memperbarui fungsi Lambda @Edge saat ini membutuhkan waktu beberapa menit. CloudFront juga membutuhkan waktu hingga 30 menit untuk menyelesaikan penyebaran perubahan ke semua titik kehadiran. Ini pada akhirnya memblokir pembaruan lebih lanjut sampai selesai.

Pola perutean header HTTP

Routing berbasis header memungkinkan Anda untuk menargetkan layanan yang benar untuk setiap permintaan dengan menentukan header HTTP dalam permintaan HTTP. Misalnya, mengirim header `x-service-a-action: get-thing` akan memungkinkan Anda untuk `get thing` dari`Service A`. Jalur permintaan masih penting, karena menawarkan panduan tentang sumber daya mana yang Anda coba kerjakan.

Selain menggunakan perutean header HTTP untuk tindakan, Anda dapat menggunakannya sebagai mekanisme untuk perutean versi, mengaktifkan flag fitur, pengujian A/B, atau kebutuhan serupa. Pada kenyataannya, Anda mungkin akan menggunakan perutean header dengan salah satu metode perutean lainnya untuk membuat yang kuat. APIs

Arsitektur untuk perutean header HTTP biasanya memiliki lapisan perutean tipis di depan layanan mikro yang merutekan ke layanan yang benar dan mengembalikan respons, seperti yang diilustrasikan dalam diagram berikut. Lapisan routing ini dapat mencakup semua layanan atau hanya beberapa layanan untuk mengaktifkan operasi seperti perutean berbasis versi.



Pro

Perubahan konfigurasi memerlukan sedikit usaha dan dapat diotomatisasi dengan mudah. Metode ini juga fleksibel dan mendukung cara-cara kreatif untuk mengekspos hanya operasi tertentu yang Anda inginkan dari layanan.

Kontra

Seperti halnya metode routing hostname, HTTP header routing mengasumsikan bahwa Anda memiliki kontrol penuh atas klien dan dapat memanipulasi header HTTP kustom. Proxy, jaringan pengiriman konten (CDNs), dan penyeimbang beban dapat membatasi ukuran header. Meskipun ini tidak mungkin menjadi masalah, ini bisa menjadi masalah tergantung pada berapa banyak header dan cookie yang Anda tambahkan.

Pola pemutus sirkuit

Niat

Pola pemutus sirkuit dapat mencegah layanan penelepon mencoba kembali panggilan ke layanan lain (callee) ketika panggilan sebelumnya menyebabkan batas waktu atau kegagalan berulang. Pola ini juga digunakan untuk mendeteksi kapan layanan callee berfungsi kembali.

Motivasi

Ketika beberapa layanan mikro berkolaborasi untuk menangani permintaan, satu atau beberapa layanan mungkin menjadi tidak tersedia atau menunjukkan latensi tinggi. Ketika aplikasi kompleks menggunakan layanan mikro, pemadaman dalam satu layanan mikro dapat menyebabkan kegagalan aplikasi. Layanan mikro berkomunikasi melalui panggilan prosedur jarak jauh, dan kesalahan sementara dapat terjadi dalam konektivitas jaringan, menyebabkan kegagalan. (Kesalahan sementara dapat ditangani dengan menggunakan coba lagi dengan pola backoff.) Selama eksekusi sinkron, berjenjang batas waktu atau kegagalan dapat menyebabkan pengalaman pengguna yang buruk.

Namun, dalam beberapa situasi, kegagalan bisa memakan waktu lebih lama untuk diselesaikan — misalnya, ketika layanan callee sedang down atau pertentangan database menghasilkan batas waktu. Dalam kasus seperti itu, jika layanan panggilan mencoba ulang panggilan berulang kali, percobaan ulang ini dapat mengakibatkan pertentangan jaringan dan konsumsi kumpulan thread database. Selain itu, jika beberapa pengguna mencoba ulang aplikasi berulang kali, ini akan memperburuk masalah dan dapat menyebabkan penurunan kinerja di seluruh aplikasi.

Pola pemutus sirkuit dipopulerkan oleh Michael Nygard dalam bukunya, Release It (Nygard 2018). Pola desain ini dapat mencegah layanan penelepon mencoba kembali panggilan layanan yang sebelumnya menyebabkan batas waktu atau kegagalan berulang. Itu juga dapat mendeteksi kapan layanan callee berfungsi kembali.

Benda pemutus sirkuit bekerja seperti pemutus sirkuit listrik yang secara otomatis mengganggu arus ketika ada kelainan pada rangkaian. Pemutus sirkuit listrik mematikan, atau tersandung, aliran arus ketika ada kesalahan. Demikian pula, objek pemutus sirkuit terletak di antara penelepon dan layanan callee, dan tersandung jika panggilan tidak tersedia.

Kekeliruan komputasi terdistribusi adalah serangkaian pernyataan yang dibuat oleh Peter Deutsch dan lainnya di Sun Microsystems. Mereka mengatakan bahwa programmer yang baru mengenal

aplikasi terdistribusi selalu membuat asumsi yang salah. Keandalan jaringan, ekspektasi latensi nol, dan keterbatasan bandwidth menghasilkan aplikasi perangkat lunak yang ditulis dengan penanganan kesalahan minimal untuk kesalahan jaringan.

Selama pemadaman jaringan, aplikasi mungkin menunggu balasan tanpa batas waktu dan terus mengkonsumsi sumber daya aplikasi. Kegagalan untuk mencoba kembali operasi ketika jaringan tersedia juga dapat menyebabkan degradasi aplikasi. Jika panggilan API ke database atau waktu layanan eksternal habis karena masalah jaringan, panggilan berulang tanpa pemutus sirkuit dapat memengaruhi biaya dan kinerja.

Penerapan

Gunakan pola ini ketika:

- Layanan penelepon membuat panggilan yang kemungkinan besar akan gagal.
- Latensi tinggi yang ditunjukkan oleh layanan callee (misalnya, ketika koneksi database lambat) menyebabkan batas waktu ke layanan pemanggil.
- Layanan penelepon membuat panggilan sinkron, tetapi layanan callee tidak tersedia atau menunjukkan latensi tinggi.

Masalah dan pertimbangan

- Implementasi agnostik layanan: Untuk mencegah pembongkaran kode, kami menyarankan Anda menerapkan objek pemutus sirkuit dengan cara agnostik layanan mikro dan berbasis API.
- Penutupan sirkuit dengan callee: Ketika callee pulih dari masalah kinerja atau kegagalan, mereka dapat memperbarui status sirkuit ke. `CLOSED` Ini adalah perpanjangan dari pola pemutus sirkuit dan dapat diimplementasikan jika tujuan waktu pemulihan Anda (RTO) membutuhkannya.
- Panggilan multithreaded: Nilai batas waktu kedaluwarsa didefinisikan sebagai periode waktu sirkuit tetap tersandung sebelum panggilan dirutekan lagi untuk memeriksa ketersediaan layanan. Ketika layanan callee dipanggil dalam beberapa utas, panggilan pertama yang gagal menentukan nilai batas waktu kedaluwarsa. Implementasi Anda harus memastikan bahwa panggilan berikutnya tidak memindahkan batas waktu kedaluwarsa tanpa henti.
- Paksa membuka atau menutup sirkuit: Administrator sistem harus memiliki kemampuan untuk membuka atau menutup sirkuit. Ini dapat dilakukan dengan memperbarui nilai batas waktu kedaluwarsa dalam tabel database.

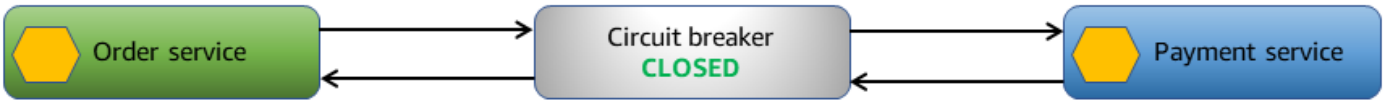
- Observabilitas: Aplikasi harus mengatur logging untuk mengidentifikasi panggilan yang gagal saat pemutus sirkuit terbuka.

Implementasi

Arsitektur tingkat tinggi

Dalam contoh berikut, penelepon adalah layanan pesanan dan yang dipanggil adalah layanan pembayaran.

Ketika tidak ada kegagalan, layanan pesanan merutekan semua panggilan ke layanan pembayaran oleh pemutus sirkuit, seperti yang ditunjukkan diagram berikut.



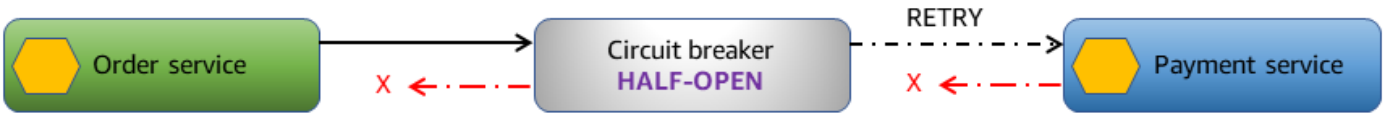
Jika waktu layanan pembayaran habis, pemutus sirkuit dapat mendeteksi batas waktu dan melacak kegagalan.

Circuit breaker with payment service failure

Jika batas waktu melebihi ambang batas yang ditentukan, aplikasi membuka sirkuit. Saat sirkuit terbuka, objek pemutus sirkuit tidak merutekan panggilan ke layanan pembayaran. Ini mengembalikan kegagalan langsung ketika layanan pesanan memanggil layanan pembayaran.

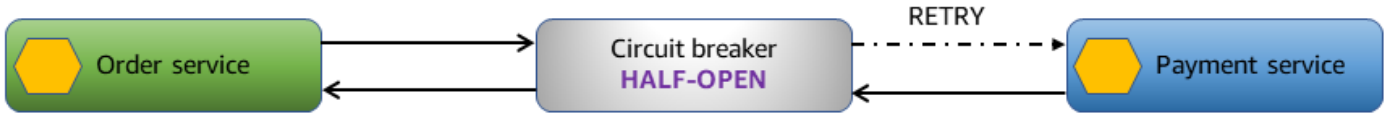
Circuit breaker stops routing to payment service

Objek pemutus sirkuit secara berkala mencoba melihat apakah panggilan ke layanan pembayaran berhasil.



Circuit breaker periodically retries payment service

Ketika panggilan ke layanan pembayaran berhasil, sirkuit ditutup, dan semua panggilan lebih lanjut diarahkan ke layanan pembayaran lagi.



Circuit breaker with working payment service

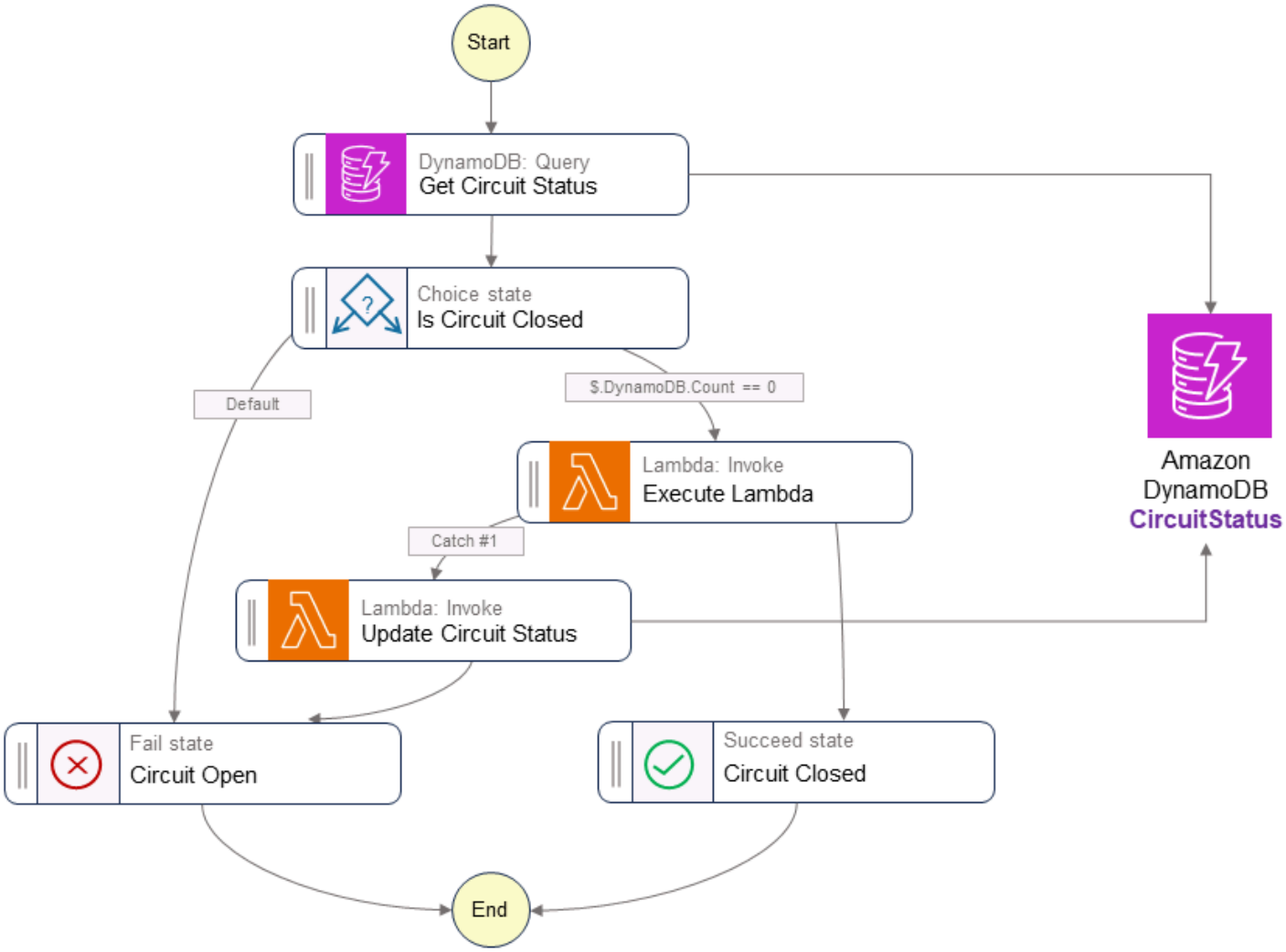
Implementasi menggunakan AWS layanan

Solusi sampel menggunakan alur kerja ekspres AWS Step Functionsuntuk mengimplementasikan pola pemutus sirkuit. Mesin status Step Functions memungkinkan Anda mengonfigurasi kemampuan coba ulang dan alur kontrol berbasis keputusan yang diperlukan untuk implementasi pola.

Solusinya juga menggunakan tabel Amazon DynamoDB sebagai penyimpanan data untuk melacak status sirkuit. Ini dapat diganti dengan datastore dalam memori seperti Amazon ElastiCache (Redis OSS) untuk kinerja yang lebih baik.

Ketika sebuah layanan ingin memanggil layanan lain, itu memulai alur kerja dengan nama layanan callee. Alur kerja mendapatkan status pemutus sirkuit dari `CircuitStatus` tabel DynamoDB, yang menyimpan layanan yang saat ini terdegradasi. Jika `CircuitStatus` berisi catatan yang belum kedaluwarsa untuk callee, sirkuit terbuka. Alur kerja Step Functions mengembalikan kegagalan langsung dan keluar dengan `FAIL` status.

Jika `CircuitStatus` tabel tidak berisi catatan untuk callee atau berisi catatan kedaluwarsa, layanan akan beroperasi. `ExecuteLambda`Langkah dalam definisi mesin status memanggil fungsi Lambda yang dikirim melalui nilai parameter. Jika panggilan berhasil, alur kerja Step Functions keluar dengan status. `SUCCESS`



Jika panggilan layanan gagal atau batas waktu terjadi, aplikasi mencoba ulang dengan backoff eksponensial untuk beberapa kali yang ditentukan. Jika panggilan layanan gagal setelah percobaan ulang, alur kerja menyisipkan catatan dalam `CircuitStatus` tabel untuk layanan dengan an`ExpiryTimeStamp`, dan alur kerja keluar dengan status. `FAIL` Panggilan berikutnya ke layanan yang sama mengembalikan kegagalan langsung selama pemutus sirkuit terbuka. `Get Circuit StatusLangkah` dalam definisi mesin negara memeriksa ketersediaan layanan berdasarkan `ExpiryTimeStamp` nilainya. Item kedaluwarsa dihapus dari `CircuitStatus` tabel dengan menggunakan fitur DynamoDB time to live (TTL).

Kode sampel

Kode berikut menggunakan fungsi `GetCircuitStatus` Lambda untuk memeriksa status pemutus sirkuit.

```
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 = "";
}
```

Kode berikut menunjukkan pernyataan Amazon States Language dalam alur kerja 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 repositori

Untuk implementasi lengkap arsitektur sampel untuk pola ini, lihat GitHub repositori di. https://github.com/aws-samples/circuit-breaker-netcore-blog

Referensi blog

- Menggunakan pola pemutus sirkuit dengan AWS Step Functions dan Amazon DynamoDB

Konten terkait

- Pola ara pencekik
- Coba lagi dengan pola backoff
- AWS App Mesh kemampuan pemutus sirkuit

Pola sumber acara

Niat

Dalam arsitektur berbasis peristiwa, pola sumber acara menyimpan peristiwa yang menghasilkan perubahan status di penyimpanan data. Ini membantu menangkap dan mempertahankan riwayat lengkap perubahan negara, dan mempromosikan auditabilitas, keterlacakan, dan kemampuan untuk menganalisis keadaan masa lalu.

Motivasi

Beberapa layanan mikro dapat berkolaborasi untuk menangani permintaan, dan mereka berkomunikasi melalui acara. Peristiwa ini dapat mengakibatkan perubahan status (data). Menyimpan objek peristiwa dalam urutan di mana mereka terjadi memberikan informasi berharga tentang keadaan saat ini dari entitas data dan informasi tambahan tentang bagaimana ia sampai pada keadaan itu.

Penerapan

Gunakan pola sumber acara saat:

- Riwayat abadi dari peristiwa yang terjadi dalam aplikasi diperlukan untuk melacak.
- Proyeksi data polyglot diperlukan dari satu sumber kebenaran (SSOT).
- Diperlukan rekonstruksi point-in-time dari status aplikasi.
- Penyimpanan jangka panjang status aplikasi tidak diperlukan, tetapi Anda mungkin ingin merekonstruksinya sesuai kebutuhan.
- Beban kerja memiliki volume baca dan tulis yang berbeda. Misalnya, Anda memiliki beban kerja intensif tulis yang tidak memerlukan pemrosesan waktu nyata.
- Change data capture (CDC) diperlukan untuk menganalisis kinerja aplikasi dan metrik lainnya.
- Data audit diperlukan untuk semua peristiwa yang terjadi dalam suatu sistem untuk tujuan pelaporan dan kepatuhan.
- Anda ingin menurunkan skenario bagaimana-jika dengan mengubah (menyisipkan, memperbarui, atau menghapus) peristiwa selama proses pemutaran ulang untuk menentukan kemungkinan status akhir.

Masalah dan pertimbangan

- Kontrol konkurensi optimis: Pola ini menyimpan setiap peristiwa yang menyebabkan perubahan status dalam sistem. Beberapa pengguna atau layanan dapat mencoba memperbarui data yang sama secara bersamaan, menyebabkan tabrakan peristiwa. Tabrakan ini terjadi ketika peristiwa yang saling bertentangan dibuat dan diterapkan pada saat yang sama, yang menghasilkan status data akhir yang tidak sesuai dengan kenyataan. Untuk mengatasi masalah ini, Anda dapat menerapkan strategi untuk mendeteksi dan menyelesaikan tabrakan peristiwa. Misalnya, Anda dapat menerapkan skema kontrol konkurensi optimis dengan menyertakan pembuatan versi atau dengan menambahkan stempel waktu ke acara untuk melacak urutan pembaruan.
- Kompleksitas: Menerapkan sumber acara memerlukan pergeseran pola pikir dari operasi CRUD tradisional ke pemikiran yang didorong oleh peristiwa. Proses replay, yang digunakan untuk mengembalikan sistem ke keadaan semula, dapat menjadi rumit untuk memastikan idempotensi data. Penyimpanan acara, cadangan, dan snapshot juga dapat menambah kompleksitas tambahan.
- Konsistensi akhir: Proyeksi data yang berasal dari peristiwa pada akhirnya konsisten karena latensi dalam memperbarui data dengan menggunakan pola pemisahan tanggung jawab permintaan perintah (CQRS) atau tampilan terwujud. Ketika konsumen memproses data dari toko acara dan penerbit mengirim data baru, proyeksi data atau objek aplikasi mungkin tidak mewakili keadaan saat ini.
- Query: Mengambil data saat ini atau agregat dari log peristiwa bisa lebih rumit dan lebih lambat dibandingkan dengan database tradisional, terutama untuk kueri kompleks dan tugas pelaporan. Untuk mengurangi masalah ini, sumber acara sering diimplementasikan dengan pola CQRS.
- Ukuran dan biaya toko acara: Toko acara dapat mengalami pertumbuhan eksponensial dalam ukuran karena peristiwa terus berlanjut, terutama dalam sistem yang memiliki throughput acara tinggi atau periode retensi yang diperpanjang. Oleh karena itu, Anda harus mengarsipkan data acara secara berkala ke penyimpanan hemat biaya untuk mencegah toko acara menjadi terlalu besar.
- Skalabilitas toko acara: Toko acara harus secara efisien menangani volume tinggi dari operasi tulis dan baca. Menskalakan toko acara bisa menjadi tantangan, jadi penting untuk memiliki penyimpanan data yang menyediakan pecahan dan partisi.
- Efisiensi dan optimasi: Pilih atau rancang toko acara yang menangani operasi tulis dan baca secara efisien. Toko acara harus dioptimalkan untuk volume acara yang diharapkan dan pola kueri untuk aplikasi. Menerapkan mekanisme pengindeksan dan kueri dapat mempercepat pengambilan peristiwa saat merekonstruksi status aplikasi. Anda juga dapat mempertimbangkan untuk

menggunakan database toko acara khusus atau pustaka yang menawarkan fitur pengoptimalan kueri.

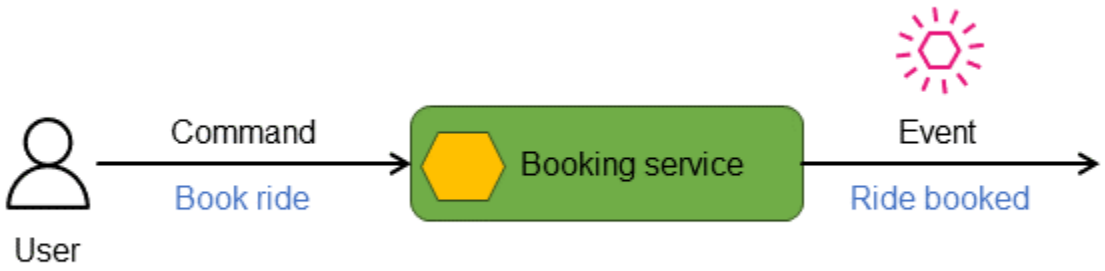
- Snapshots: Anda harus mencadangkan log peristiwa secara berkala dengan aktivasi berbasis waktu. Memutar ulang peristiwa pada cadangan data yang berhasil terakhir diketahui akan mengarah pada point-in-time pemulihan status aplikasi. Tujuan titik pemulihan (RPO) adalah jumlah waktu maksimum yang dapat diterima sejak titik pemulihan data terakhir. RPO menentukan apa yang dianggap sebagai hilangnya data yang dapat diterima antara titik pemulihan terakhir dan gangguan layanan. Frekuensi snapshot harian data dan toko acara harus didasarkan pada RPO untuk aplikasi.
- Sensitivitas waktu: Peristiwa disimpan dalam urutan terjadinya. Oleh karena itu, keandalan jaringan merupakan faktor penting untuk dipertimbangkan ketika Anda menerapkan pola ini. Masalah latensi dapat menyebabkan status sistem yang salah. Gunakan antrian first in, first out (FIFO) dengan at-most-once pengiriman untuk membawa acara ke toko acara.
- Kinerja pemutaran ulang acara: Memutar ulang sejumlah besar peristiwa untuk merekonstruksi status aplikasi saat ini dapat memakan waktu. Upaya optimasi diperlukan untuk meningkatkan kinerja, terutama saat memutar ulang peristiwa dari data yang diarsipkan.
- Pembaruan sistem eksternal: Aplikasi yang menggunakan pola sumber acara dapat memperbarui penyimpanan data di sistem eksternal, dan mungkin menangkap pembaruan ini sebagai objek peristiwa. Selama tayangan ulang acara, ini mungkin menjadi masalah jika sistem eksternal tidak mengharapkan pembaruan. Dalam kasus seperti itu, Anda dapat menggunakan bendera fitur untuk mengontrol pembaruan sistem eksternal.
- Kueri sistem eksternal: Ketika panggilan sistem eksternal sensitif terhadap tanggal dan waktu panggilan, data yang diterima dapat disimpan di penyimpanan data internal untuk digunakan selama pemutaran ulang.
- Versi acara: Saat aplikasi berkembang, struktur peristiwa (skema) dapat berubah. Menerapkan strategi pembuatan versi untuk acara untuk memastikan kompatibilitas mundur dan maju diperlukan. Ini dapat melibatkan termasuk bidang versi dalam payload acara dan menangani versi acara yang berbeda dengan tepat selama pemutaran ulang.

Implementasi

Arsitektur tingkat tinggi

Perintah dan acara

Dalam aplikasi layanan mikro berbasis peristiwa terdistribusi, perintah mewakili instruksi atau permintaan yang dikirim ke layanan, biasanya dengan maksud memulai perubahan statusnya. Layanan memproses perintah ini dan mengevaluasi validitas dan penerapan perintah ke keadaan saat ini. Jika perintah berjalan dengan sukses, layanan merespons dengan memancarkan peristiwa yang menandakan tindakan yang diambil dan informasi status yang relevan. Misalnya, dalam diagram berikut, layanan pemesanan merespons perintah Book ride dengan memancarkan acara yang dipesan Ride.



Toko acara

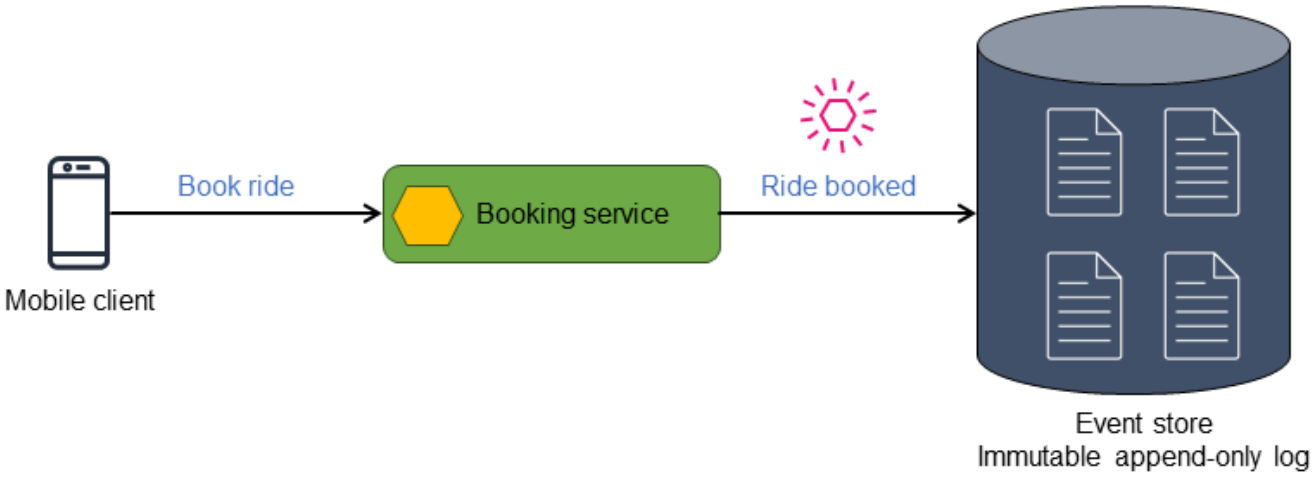
Peristiwa masuk ke repositori atau penyimpanan data yang tidak dapat diubah, hanya ditambahkan, diurutkan secara kronologis yang dikenal sebagai toko acara. Setiap perubahan status diperlakukan sebagai objek peristiwa individu. Objek entitas atau penyimpanan data dengan keadaan awal yang diketahui, keadaan saat ini, dan point-in-time tampilan apa pun dapat direkonstruksi dengan memutar ulang peristiwa dalam urutan kemunculannya.

Toko acara bertindak sebagai catatan sejarah dari semua tindakan dan perubahan negara, dan berfungsi sebagai sumber kebenaran tunggal yang berharga. Anda dapat menggunakan toko acara untuk mendapatkan up-to-date status akhir sistem dengan meneruskan peristiwa melalui prosesor replay, yang menerapkan peristiwa ini untuk menghasilkan representasi akurat dari status sistem terbaru. Anda juga dapat menggunakan toko acara untuk menghasilkan point-in-time perspektif status dengan memutar ulang peristiwa melalui prosesor replay. Dalam pola sumber peristiwa, status saat ini mungkin tidak sepenuhnya diwakili oleh objek peristiwa terbaru. Anda dapat memperoleh keadaan saat ini dengan salah satu dari tiga cara:

- Dengan menggabungkan peristiwa terkait. Objek peristiwa terkait digabungkan untuk menghasilkan status saat ini untuk kueri. Pendekatan ini sering digunakan bersama dengan pola CQRS, di mana peristiwa digabungkan dan ditulis ke dalam penyimpanan data hanya-baca.

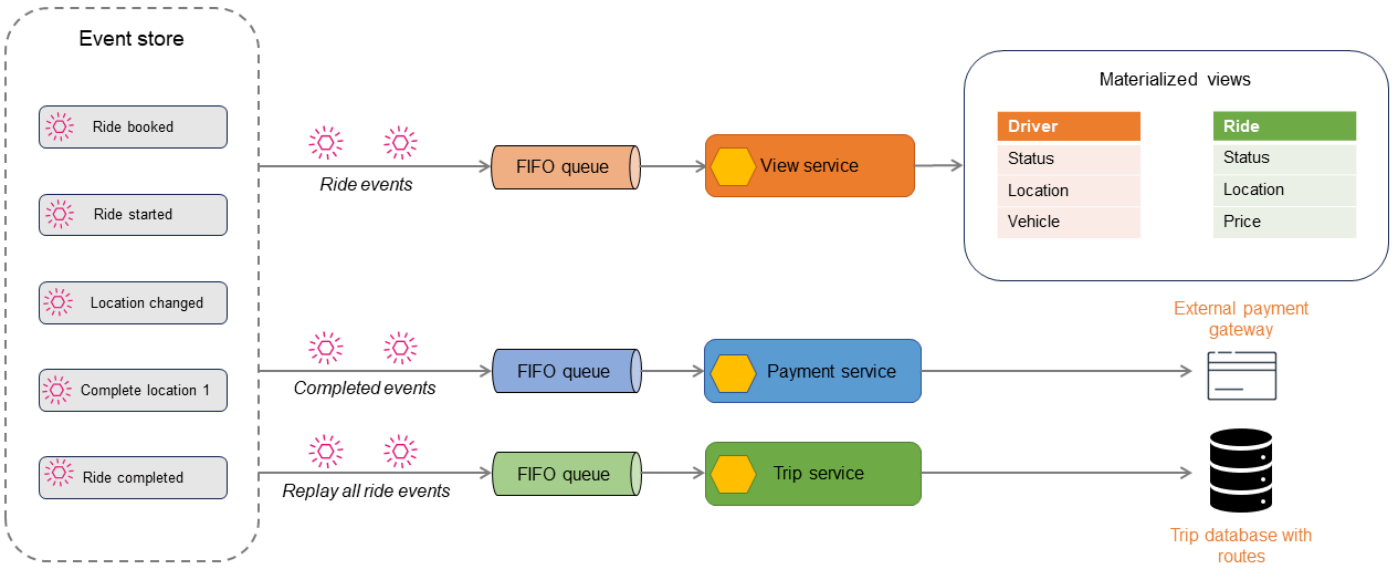
- Dengan menggunakan tampilan yang terwujud. Anda dapat menggunakan sumber peristiwa dengan pola tampilan terwujud untuk menghitung atau meringkas data peristiwa dan mendapatkan status data terkait saat ini.
- Dengan memutar ulang acara. Objek acara dapat diputar ulang untuk melakukan tindakan untuk menghasilkan keadaan saat ini.

Diagram berikut menunjukkan `Ride booked` acara yang disimpan di toko acara.



Toko acara menerbitkan acara yang disimpannya, dan acara dapat disaring dan diarahkan ke prosesor yang sesuai untuk tindakan selanjutnya. Misalnya, peristiwa dapat dirutekan ke prosesor tampilan yang merangkum status dan menunjukkan tampilan yang terwujud. Peristiwa diubah ke format data penyimpanan data target. Arsitektur ini dapat diperluas untuk memperoleh berbagai jenis penyimpanan data, yang mengarah pada persistensi data polyglot.

Diagram berikut menjelaskan peristiwa dalam aplikasi pemesanan perjalanan. Semua peristiwa yang terjadi dalam aplikasi disimpan di toko acara. Peristiwa yang disimpan kemudian disaring dan diarahkan ke konsumen yang berbeda.



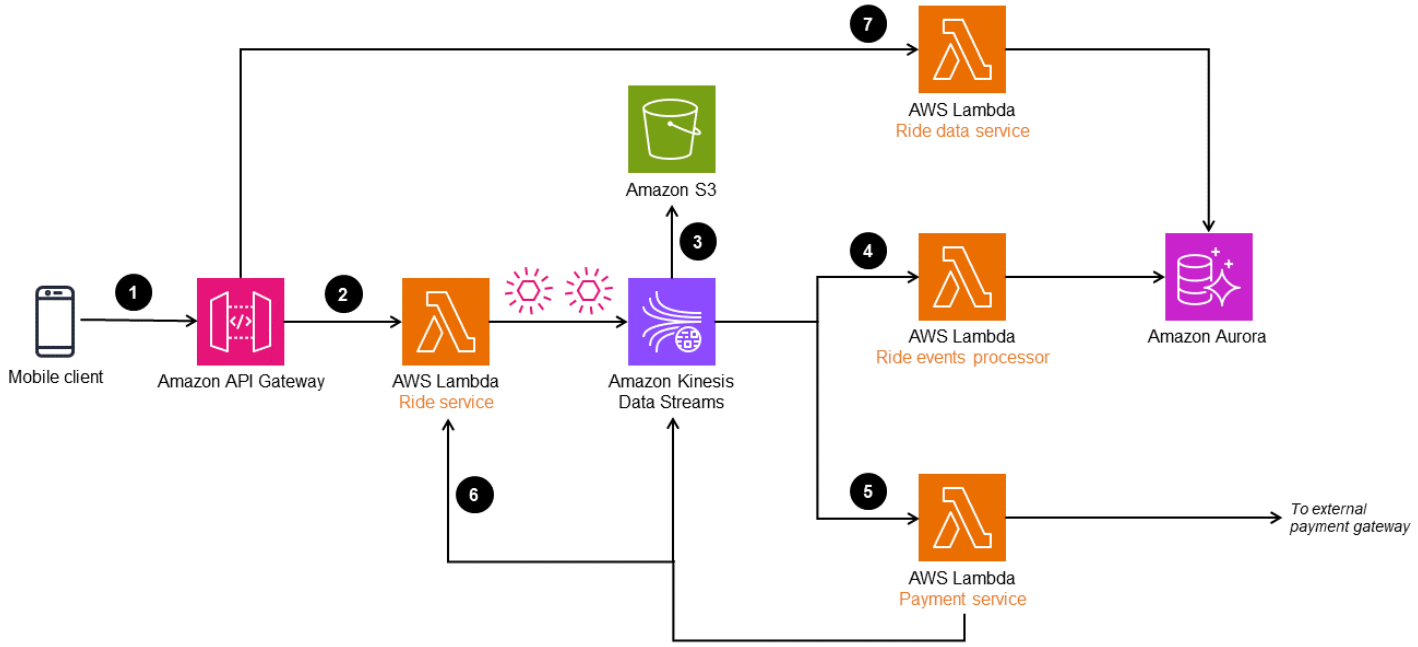
Peristiwa perjalanan dapat digunakan untuk menghasilkan penyimpanan data hanya-baca dengan menggunakan CQRS atau pola tampilan terwujud. Anda dapat memperoleh status perjalanan saat ini, pengemudi, atau pemesanan dengan menanyakan toko baca. Beberapa peristiwa, seperti `Location changed` atau`Ride completed`, dipublikasikan ke konsumen lain untuk diproses pembayaran. Ketika perjalanan selesai, semua acara perjalanan diputar ulang untuk membangun riwayat perjalanan untuk tujuan audit atau pelaporan.

Pola sumber peristiwa sering digunakan dalam aplikasi yang memerlukan point-in-time pemulihan, dan juga ketika data harus diproyeksikan dalam format yang berbeda dengan menggunakan satu sumber kebenaran. Kedua operasi ini memerlukan proses pemutaran ulang untuk menjalankan acara dan mendapatkan status akhir yang diperlukan. Prosesor replay mungkin juga memerlukan titik awal yang diketahui — idealnya bukan dari peluncuran aplikasi, karena itu tidak akan menjadi proses yang efisien. Kami menyarankan Anda mengambil snapshot reguler dari status sistem dan menerapkan sejumlah kecil peristiwa untuk mendapatkan status. up-to-date

Implementasi menggunakan layanan AWS

Dalam arsitektur berikut, Amazon Kinesis Data Streams digunakan sebagai toko acara. Layanan ini menangkap dan mengelola perubahan aplikasi sebagai peristiwa, dan menawarkan solusi streaming data throughput tinggi dan real-time. Untuk menerapkan pola sumber acara di AWS, Anda juga dapat menggunakan layanan seperti Amazon EventBridge dan Amazon Managed Streaming for Apache Kafka (Amazon MSK) berdasarkan kebutuhan aplikasi Anda.

Untuk meningkatkan daya tahan dan mengaktifkan audit, Anda dapat mengarsipkan peristiwa yang ditangkap oleh Kinesis Data Streams di Amazon Simple Storage Service (Amazon S3). Pendekatan penyimpanan ganda ini membantu menyimpan data peristiwa historis dengan aman untuk tujuan analisis dan kepatuhan di masa mendatang.



Alur kerja terdiri dari langkah-langkah berikut:

1. Permintaan pemesanan perjalanan dilakukan melalui klien seluler ke titik akhir Amazon API Gateway.
2. Layanan mikro perjalanan (fungsi `Ride service` Lambda) menerima permintaan, mengubah objek, dan menerbitkan ke Kinesis Data Streams.
3. Data peristiwa di Kinesis Data Streams disimpan di Amazon S3 untuk tujuan kepatuhan dan riwayat audit.
4. Peristiwa diubah dan diproses oleh fungsi `Ride event processor` Lambda dan disimpan dalam database Amazon Aurora untuk memberikan tampilan terwujud untuk data perjalanan.
5. Acara perjalanan yang telah selesai disaring dan dikirim untuk diproses pembayaran ke gateway pembayaran eksternal. Ketika pembayaran selesai, acara lain dikirim ke Kinesis Data Streams untuk memperbarui database Ride.
6. Ketika perjalanan selesai, acara perjalanan diputar ulang ke fungsi `Ride service` Lambda untuk membangun rute dan sejarah perjalanan.

7. Informasi perjalanan dapat dibaca melaluiRide data service, yang dibaca dari database Aurora.

API Gateway juga dapat mengirim objek acara langsung ke Kinesis Data Streams tanpa fungsi LambdaRide service. Namun, dalam sistem yang kompleks seperti layanan ride hailing, objek acara mungkin perlu diproses dan diperkaya sebelum tertelan ke dalam aliran data. Untuk alasan ini, arsitektur memiliki Ride service yang memproses peristiwa sebelum mengirimnya ke Kinesis Data Streams.

Referensi blog

- Baru untuk AWS Lambda — SQS FIFO sebagai sumber acara

Pola arsitektur heksagonal

Niat

Pola arsitektur heksagonal, yang juga dikenal sebagai pola port dan adaptor, diusulkan oleh Dr. Alistair Cockburn pada tahun 2005. Ini bertujuan untuk membuat arsitektur yang digabungkan secara longgar di mana komponen aplikasi dapat diuji secara independen, tanpa ketergantungan pada penyimpanan data atau antarmuka pengguna (). UIs Pola ini membantu mencegah penguncian teknologi penyimpanan data dan. UIs Ini membuatnya lebih mudah untuk mengubah tumpukan teknologi dari waktu ke waktu, dengan dampak terbatas atau tidak ada pada logika bisnis. Dalam arsitektur yang digabungkan secara longgar ini, aplikasi berkomunikasi dengan komponen eksternal melalui antarmuka yang disebut port, dan menggunakan adaptor untuk menerjemahkan pertukaran teknis dengan komponen ini.

Motivasi

Pola arsitektur heksagonal digunakan untuk mengisolasi logika bisnis (logika domain) dari kode infrastruktur terkait, seperti kode untuk mengakses database atau eksternal. APIs Pola ini berguna untuk membuat logika bisnis dan kode infrastruktur yang digabungkan secara longgar untuk AWS Lambda fungsi yang memerlukan integrasi dengan layanan eksternal. Dalam arsitektur tradisional, praktik umum adalah menanamkan logika bisnis di lapisan database sebagai prosedur tersimpan dan di antarmuka pengguna. Praktik ini, bersama dengan menggunakan konstruksi khusus UI dalam logika bisnis, mengarah pada arsitektur yang digabungkan erat yang menyebabkan kemacetan dalam migrasi database dan upaya modernisasi pengalaman pengguna (UX). Pola arsitektur heksagonal memungkinkan Anda untuk merancang sistem dan aplikasi Anda dengan tujuan bukan oleh teknologi. Strategi ini menghasilkan komponen aplikasi yang mudah ditukar seperti database, UX, dan komponen layanan.

Penerapan

Gunakan pola arsitektur heksagonal saat:

- Anda ingin memisahkan arsitektur aplikasi Anda untuk membuat komponen yang dapat diuji sepenuhnya.
- Beberapa jenis klien dapat menggunakan logika domain yang sama.

- Komponen UI dan database Anda memerlukan penyegaran teknologi berkala yang tidak memengaruhi logika aplikasi.
- Aplikasi Anda memerlukan beberapa penyedia input dan konsumen keluaran, dan menyesuaikan logika aplikasi menyebabkan kompleksitas kode dan kurangnya ekstensibilitas.

Masalah dan pertimbangan

- Desain berbasis domain: Arsitektur heksagonal bekerja sangat baik dengan desain berbasis domain (DDD). Setiap komponen aplikasi mewakili sub-domain dalam DDD, dan arsitektur heksagonal dapat digunakan untuk mencapai kopling longgar di antara komponen aplikasi.
- Testability: Dengan desain, arsitektur heksagonal menggunakan abstraksi untuk input dan output. Oleh karena itu, menulis pengujian unit dan pengujian secara terpisah menjadi lebih mudah karena kopling longgar yang melekat.
- Kompleksitas: Kompleksitas memisahkan logika bisnis dari kode infrastruktur, ketika ditangani dengan hati-hati, dapat membawa manfaat besar seperti kelincahan, cakupan pengujian, dan kemampuan beradaptasi teknologi. Jika tidak, masalah bisa menjadi rumit untuk dipecahkan.
- Overhead pemeliharaan: Kode adaptor tambahan yang membuat arsitektur dapat dicolokkan dibenarkan hanya jika komponen aplikasi memerlukan beberapa sumber input dan tujuan output untuk menulis, atau ketika penyimpanan data input dan output harus berubah seiring waktu. Jika tidak, adaptor menjadi lapisan tambahan lain untuk dipelihara, yang memperkenalkan overhead pemeliharaan.
- Masalah latensi: Menggunakan port dan adaptor menambahkan lapisan lain, yang dapat mengakibatkan latensi.

Implementasi

Arsitektur heksagonal mendukung isolasi aplikasi dan logika bisnis dari kode infrastruktur dan dari kode yang mengintegrasikan aplikasi dengan UIs, eksternal, database APIs, dan broker pesan. Anda dapat dengan mudah menghubungkan komponen logika bisnis ke komponen lain (seperti database) dalam arsitektur aplikasi melalui port dan adaptor.

Port adalah titik masuk teknologi-agnostik ke dalam komponen aplikasi. Antarmuka khusus ini menentukan antarmuka yang memungkinkan aktor eksternal untuk berkomunikasi dengan komponen aplikasi, terlepas dari siapa atau apa yang mengimplementasikan antarmuka. Ini mirip dengan

bagaimana port USB memungkinkan berbagai jenis perangkat untuk berkomunikasi dengan komputer, selama mereka menggunakan adaptor USB.

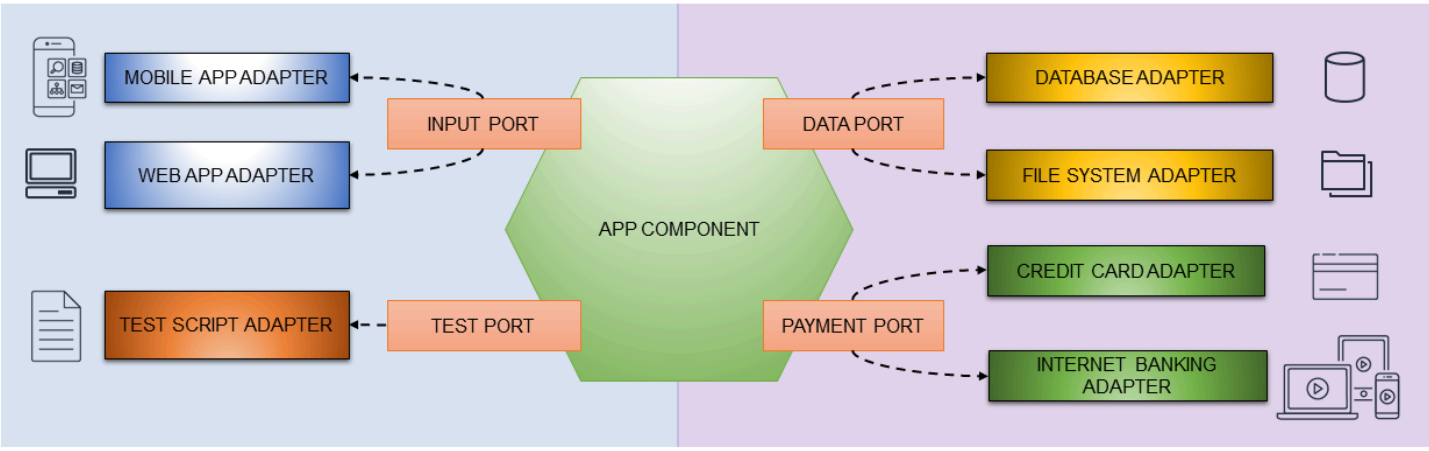
Adaptor berinteraksi dengan aplikasi melalui port dengan menggunakan teknologi tertentu. Adaptor dicolokkan ke port ini, menerima data dari atau memberikan data ke port, dan mengubah data untuk diproses lebih lanjut. Misalnya, adaptor REST memungkinkan aktor untuk berkomunikasi dengan komponen aplikasi melalui REST API. Sebuah port dapat memiliki beberapa adaptor tanpa risiko ke port atau komponen aplikasi. Untuk memperluas contoh sebelumnya, menambahkan adaptor GraphQL ke port yang sama menyediakan sarana tambahan bagi aktor untuk berinteraksi dengan aplikasi melalui GraphQL API tanpa mempengaruhi REST API, port, atau aplikasi.

Port terhubung ke aplikasi, dan adaptor berfungsi sebagai koneksi ke dunia luar. Anda dapat menggunakan port untuk membuat komponen aplikasi yang digabungkan secara longgar, dan bertukar komponen dependen dengan mengubah adaptor. Hal ini memungkinkan komponen aplikasi untuk berinteraksi dengan input dan output eksternal tanpa perlu memiliki kesadaran kontekstual. Komponen dapat ditukar di tingkat mana pun, yang memfasilitasi pengujian otomatis. Anda dapat menguji komponen secara independen tanpa dependensi apa pun pada kode infrastruktur alih-alih menyediakan seluruh lingkungan untuk melakukan pengujian. Logika aplikasi tidak bergantung pada faktor eksternal, jadi pengujian disederhanakan dan menjadi lebih mudah untuk mengejek dependensi.

Misalnya, dalam arsitektur yang digabungkan secara longgar, komponen aplikasi harus dapat membaca dan menulis data tanpa mengetahui detail penyimpanan data. Tanggung jawab komponen aplikasi adalah memasok data ke antarmuka (port). Adaptor mendefinisikan logika penulisan ke penyimpanan data, yang dapat berupa database, sistem file, atau sistem penyimpanan objek seperti Amazon S3, tergantung pada kebutuhan aplikasi.

Arsitektur tingkat tinggi

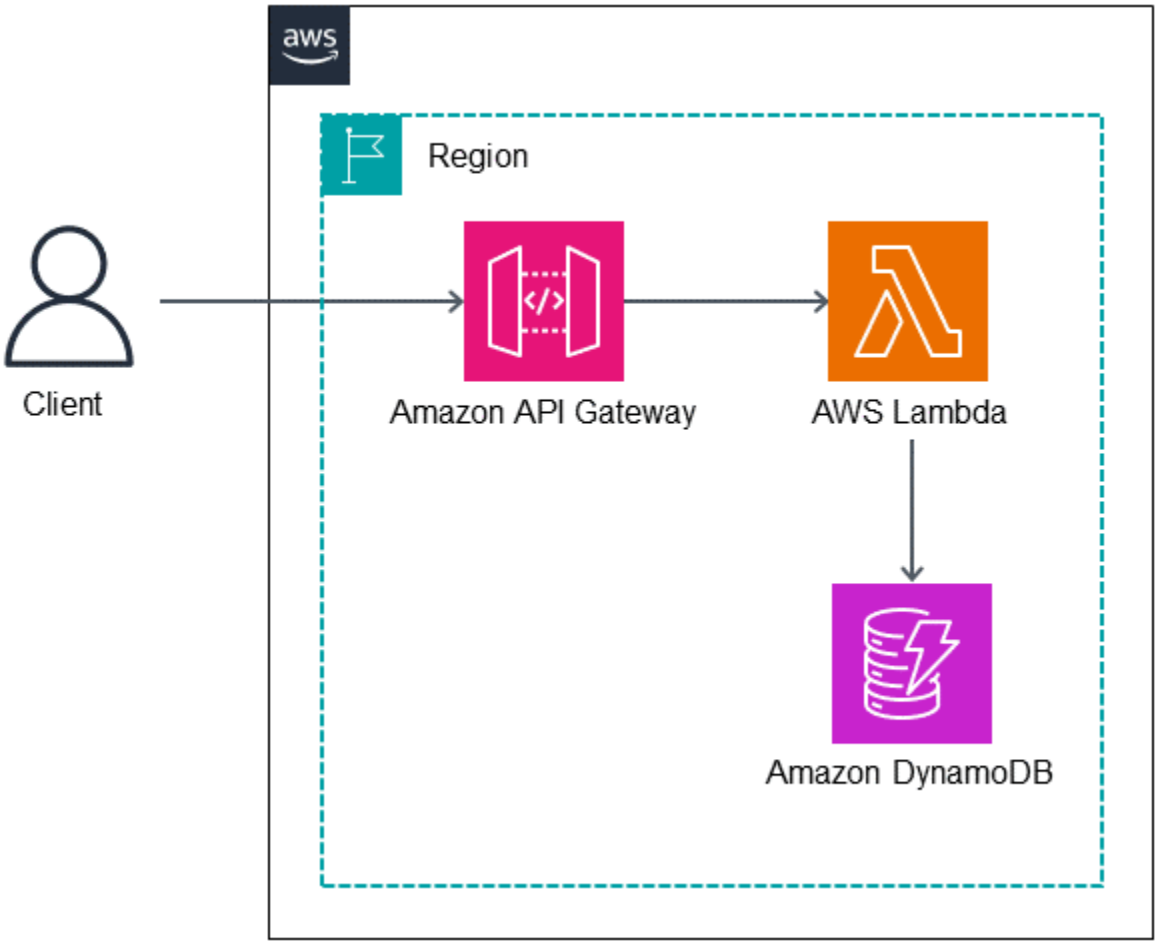
Komponen aplikasi atau aplikasi berisi logika bisnis inti. Ini menerima perintah atau pertanyaan dari port, dan mengirimkan permintaan keluar melalui port ke aktor eksternal, yang diimplementasikan melalui adaptor, seperti yang diilustrasikan dalam diagram berikut.



Implementasi menggunakan Layanan AWS

AWS Lambda fungsi sering berisi logika bisnis dan kode integrasi database, yang digabungkan erat untuk memenuhi tujuan. Anda dapat menggunakan pola arsitektur heksagonal untuk memisahkan logika bisnis dari kode infrastruktur. Pemisahan ini memungkinkan pengujian unit logika bisnis tanpa ketergantungan pada kode database, dan meningkatkan kelincahan proses pengembangan.

Dalam arsitektur berikut, fungsi Lambda mengimplementasikan pola arsitektur heksagonal. Fungsi Lambda diprakarsai oleh Amazon API Gateway REST API. Fungsi mengimplementasikan logika bisnis dan menulis data ke tabel DynamoDB.



Kode sampel

Kode sampel di bagian ini menunjukkan cara menerapkan model domain dengan menggunakan Lambda, memisahkannya dari kode infrastruktur (seperti kode untuk mengakses DynamoDB), dan menerapkan pengujian unit untuk fungsi tersebut.

Model domain

Kelas model domain tidak memiliki pengetahuan tentang komponen eksternal atau dependensi — hanya mengimplementasikan logika bisnis. Dalam contoh berikut, kelas `Recipient` adalah kelas model domain yang memeriksa tumpang tindih pada tanggal reservasi.

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

Port masukan

`RecipientInputPort`Kelas terhubung ke kelas penerima dan menjalankan logika domain.

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

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

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

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

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

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

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

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

Kelas adaptor DynamoDB

`DDBRecipientAdapter`Kelas mengimplementasikan akses ke tabel 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": []
            }
          # ...
```

Fungsi Lambda `get_recipient_input_port` adalah pabrik untuk instance kelas. `RecipientInputPort` Ini membangun contoh kelas port output dengan instance adaptor terkait.

```
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
        }),
    }
```

Pengujian unit

Anda dapat menguji logika bisnis untuk kelas model domain dengan menyuntikkan kelas tiruan. Contoh berikut menyediakan unit test untuk `Recipent` kelas model domain.

```
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 repositori

Untuk implementasi lengkap arsitektur sampel untuk pola ini, lihat GitHub repositori di https://github.com/aws-samples/ aws-lambda-domain-model -sample.

Konten terkait

- Arsitektur heksagonal, artikel oleh Alistair Cockburn
- Mengembangkan arsitektur evolusioner dengan AWS Lambda (posting AWS blog dalam bahasa Jepang)

Video

Video berikut (dalam bahasa Jepang) membahas penggunaan arsitektur heksagonal dalam implementasi model domain dengan menggunakan fungsi Lambda.

Pola terbitkan-berlangganan

Niat

Pola berlangganan publikasi, yang juga dikenal sebagai pola pub-sub, adalah pola pesan yang memisahkan pengirim pesan (penerbit) dari penerima yang tertarik (pelanggan). Pola ini mengimplementasikan komunikasi asinkron dengan menerbitkan pesan atau peristiwa melalui perantara yang dikenal sebagai broker pesan atau router (infrastruktur pesan). Pola publish-subscribe meningkatkan skalabilitas dan responsif bagi pengirim dengan menurunkan tanggung jawab pengiriman pesan ke infrastruktur pesan, sehingga pengirim dapat fokus pada pemrosesan pesan inti.

Motivasi

Dalam arsitektur terdistribusi, komponen sistem sering perlu memberikan informasi ke komponen lain karena peristiwa terjadi di dalam sistem. Pola berlangganan publikasi memisahkan kekhawatiran sehingga aplikasi dapat fokus pada kemampuan inti mereka sementara infrastruktur pesan menangani tanggung jawab komunikasi seperti perutean pesan dan pengiriman yang andal. Pola berlangganan publikasi memungkinkan pesan asinkron untuk memisahkan penerbit dan pelanggan. Penerbit juga dapat mengirim pesan tanpa sepengetahuan pelanggan.

Penerapan

Gunakan pola berlangganan publikasi saat:

- Pemrosesan paralel diperlukan jika satu pesan memiliki alur kerja yang berbeda.
- Menyiarkan pesan ke beberapa pelanggan dan tanggapan real-time dari penerima tidak diperlukan.
- Sistem atau aplikasi dapat mentolerir konsistensi akhirnya untuk data atau status.
- Aplikasi atau komponen harus berkomunikasi dengan aplikasi atau layanan lain yang mungkin menggunakan bahasa, protokol, atau platform yang berbeda.

Masalah dan pertimbangan

- Ketersediaan pelanggan: Penerbit tidak mengetahui apakah pelanggan mendengarkan, dan mereka mungkin tidak. Pesan yang dipublikasikan bersifat sementara dan dapat mengakibatkan dijatuhkan jika pelanggan tidak tersedia.
- Jaminan pengiriman pesan: Biasanya, pola berlangganan publikasi tidak dapat menjamin pengiriman pesan ke semua jenis pelanggan, meskipun layanan tertentu seperti Amazon Simple Notification Service (Amazon SNS) dapat memberikan pengiriman tepat sekali ke beberapa subset pelanggan.
- Time to live (TTL): Pesan memiliki masa pakai dan kedaluwarsa jika tidak diproses dalam jangka waktu tersebut. Pertimbangkan untuk menambahkan pesan yang dipublikasikan ke antrian sehingga dapat bertahan, dan menjamin pemrosesan di luar periode TTL.
- Relevansi pesan: Produsen dapat menetapkan rentang waktu untuk relevansi sebagai bagian dari data pesan, dan pesan dapat dibuang setelah tanggal ini. Pertimbangkan merancang konsumen untuk memeriksa informasi ini sebelum Anda memutuskan bagaimana memproses pesan.
- Konsistensi akhir: Ada penundaan antara waktu pesan diterbitkan dan waktu yang dikonsumsi oleh pelanggan. Hal ini dapat mengakibatkan penyimpanan data pelanggan menjadi konsisten ketika konsistensi yang kuat diperlukan. Konsistensi akhirnya mungkin juga menjadi masalah ketika produsen dan konsumen membutuhkan interaksi yang hampir real time.
- Komunikasi searah: Pola berlangganan penerbitan dianggap searah. Aplikasi yang memerlukan pesan dua arah dengan saluran langganan kembali harus mempertimbangkan untuk menggunakan pola permintaan-balasan jika diperlukan respons sinkron.
- Urutan pesan: Pemesanan pesan tidak dijamin. Jika konsumen memerlukan pesan yang dipesan, kami sarankan Anda menggunakan topik Amazon SNS FIFO untuk menjamin pemesanan.
- Duplikasi pesan: Berdasarkan infrastruktur pesan, pesan duplikat dapat dikirimkan ke konsumen. Konsumen harus dirancang untuk menjadi idempoten untuk menangani pemrosesan pesan duplikat. Atau, gunakan topik Amazon SNS FIFO untuk menjamin pengiriman tepat sekali.
- Pemfilteran pesan: Konsumen seringkali hanya tertarik pada subset pesan yang diterbitkan oleh produser. Menyediakan mekanisme untuk memungkinkan pelanggan memfilter atau mempersempit pesan yang mereka terima dengan menyediakan topik atau filter konten.
- Pemutaran ulang pesan: Kemampuan pemutaran ulang pesan mungkin bergantung pada infrastruktur pesan. Anda juga dapat memberikan implementasi khusus tergantung pada kasus penggunaan.

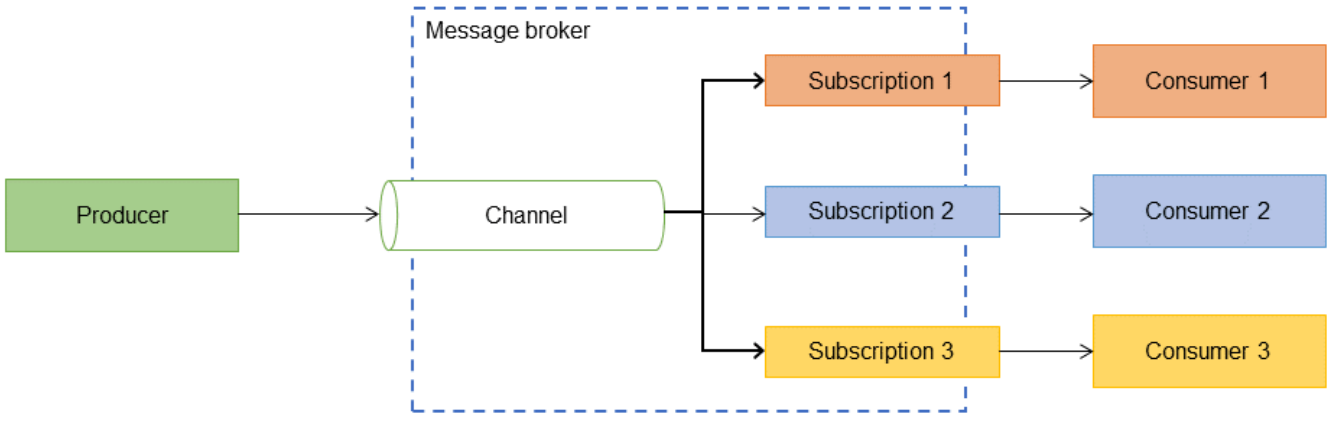
- Antrian surat mati: Dalam sistem pos, kantor surat mati adalah fasilitas untuk memproses surat yang tidak terkirim. Di pub/sub messaging, antrian dead-letter (DLQ) adalah antrian untuk pesan yang tidak dapat dikirim ke titik akhir berlangganan.

Implementasi

Arsitektur tingkat tinggi

Dalam pola berlangganan publikasi, subsistem pesan asinkron yang dikenal sebagai broker pesan atau router melacak langganan. Ketika produser menerbitkan sebuah acara, infrastruktur pesan mengirimkan pesan ke setiap konsumen. Setelah pesan dikirim ke pelanggan, pesan tersebut dihapus dari infrastruktur pesan sehingga tidak dapat diputar ulang, dan pelanggan baru tidak melihat acara tersebut. Broker pesan atau router memisahkan produsen acara dari konsumen pesan dengan:

- Menyediakan saluran input bagi produser untuk mempublikasikan peristiwa yang dikemas ke dalam pesan, menggunakan format pesan yang ditentukan.
- Membuat saluran keluaran individual per langganan. Langganan adalah koneksi konsumen, di mana mereka mendengarkan pesan acara yang terkait dengan saluran input tertentu.
- Menyalin pesan dari saluran input ke saluran output untuk semua konsumen saat acara dipublikasikan.



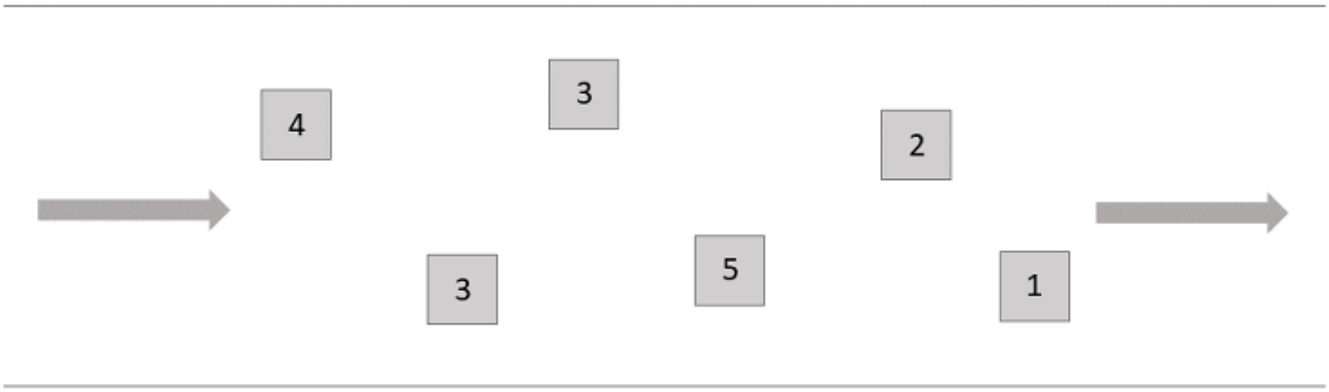
Implementasi menggunakan layanan AWS

Amazon SNS

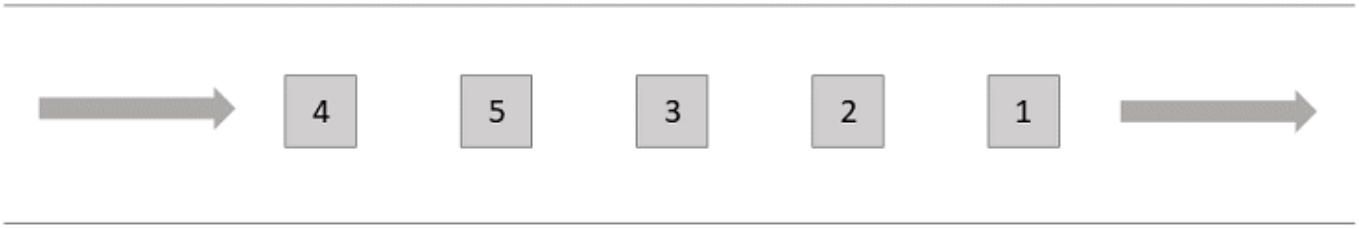
Amazon SNS adalah layanan penerbit-pelanggan yang dikelola sepenuhnya yang menyediakan pesan application-to-application (A2A) untuk memisahkan aplikasi terdistribusi. Ini juga menyediakan pesan application-to-person (A2P) untuk mengirim SMS, email, dan pemberitahuan push lainnya.

Amazon SNS menyediakan dua jenis topik: standar dan first in, first out (FIFO).

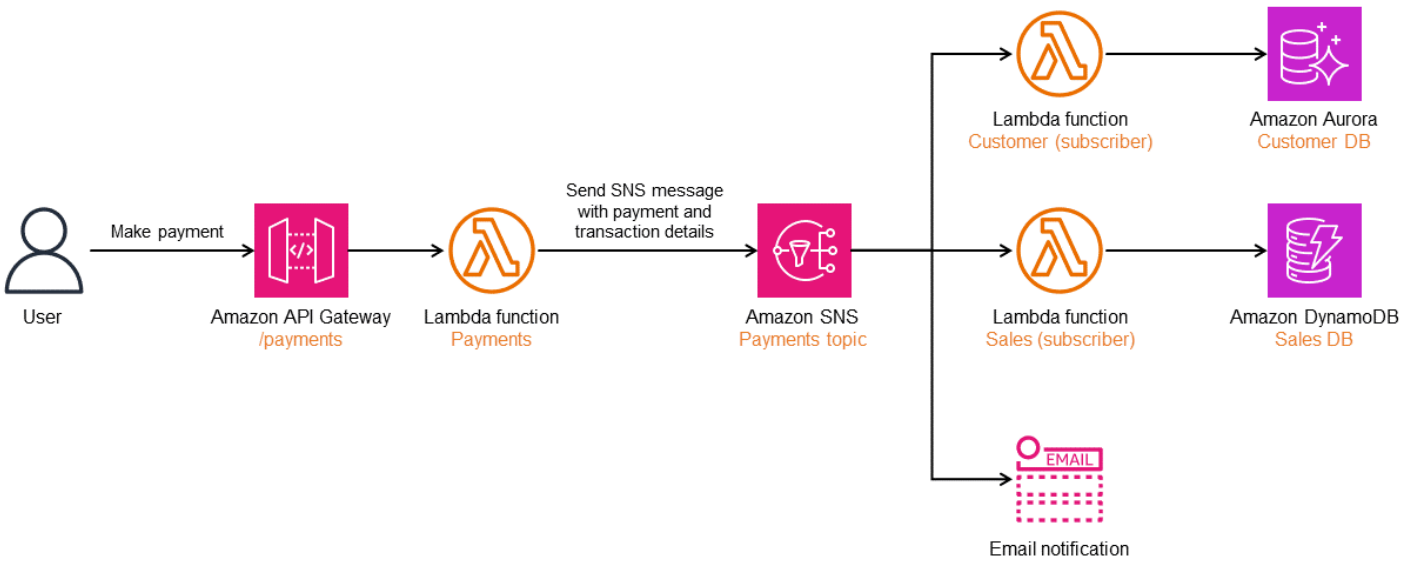
- Topik standar mendukung jumlah pesan yang tidak terbatas per detik, dan memberikan pemesanan dan deduplikasi upaya terbaik.



- Topik FIFO menyediakan urutan dan deduplikasi yang ketat, dan mendukung hingga 300 pesan per detik atau 10 MB per detik per topik FIFO (mana yang lebih dulu).

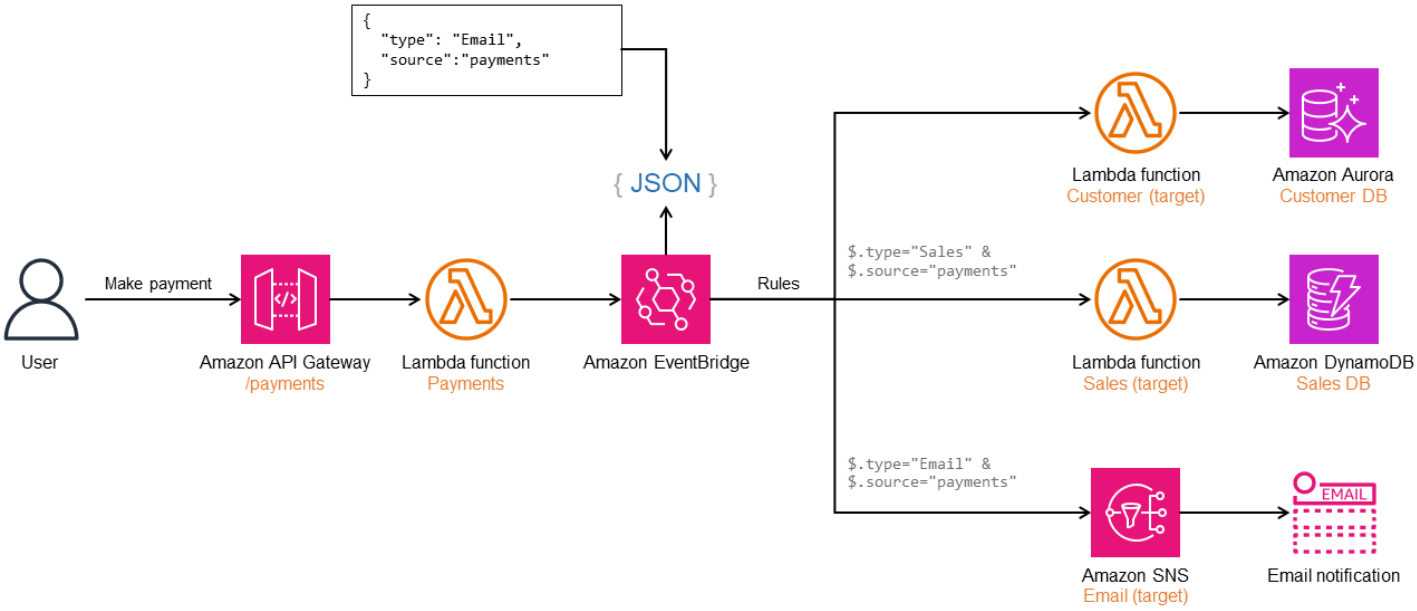


Ilustrasi berikut menunjukkan bagaimana Anda dapat menggunakan Amazon SNS untuk menerapkan pola berlangganan publikasi. Setelah pengguna melakukan pembayaran, pesan SNS dikirim oleh fungsi `Payments` Lambda ke `Payments` topik SNS. Topik SNS ini memiliki tiga pelanggan. Setiap pelanggan menerima salinan pesan dan memprosesnya.



Amazon EventBridge

Anda dapat menggunakan Amazon EventBridge saat Anda memerlukan perutean pesan yang lebih kompleks dari beberapa produsen di berbagai protokol ke konsumen berlangganan, atau langganan langsung dan penggemar. EventBridge juga mendukung routing berbasis konten, penyaringan, pengurutan, dan pemisahan atau agregasi. Dalam ilustrasi berikut, EventBridge digunakan untuk membangun versi pola berlangganan publikasi di mana pelanggan ditentukan dengan menggunakan aturan acara. Setelah pengguna melakukan pembayaran, fungsi `Payments` Lambda mengirimkan pesan ke EventBridge dengan menggunakan bus acara default berdasarkan skema kustom yang memiliki tiga aturan yang menunjuk ke target yang berbeda. Setiap layanan mikro memproses pesan dan melakukan tindakan yang diperlukan.



Lokakarya

- Membangun arsitektur berbasis peristiwa di AWS
- Kirim Pemberitahuan Acara Fanout dengan Amazon Simple Queue Service (Amazon SQS) dan Amazon Simple Notification Service (Amazon SNS)

Referensi blog

- Memilih antara layanan pesan untuk aplikasi tanpa server
- Merancang aplikasi tanpa server yang tahan lama dengan DLQs Amazon SNS, Amazon SQS, AWS Lambda
- Sederhanakan pesan pub/sub Anda dengan pemfilteran pesan Amazon SNS

Konten terkait

- Fitur pesan pub/sub

Coba lagi dengan pola backoff

Niat

Coba lagi dengan pola backoff meningkatkan stabilitas aplikasi dengan mencoba kembali operasi secara transparan yang gagal karena kesalahan sementara.

Motivasi

Dalam arsitektur terdistribusi, kesalahan sementara mungkin disebabkan oleh pelambatan layanan, hilangnya konektivitas jaringan sementara, atau tidak tersedianya layanan sementara. Secara otomatis mencoba kembali operasi yang gagal karena kesalahan sementara ini meningkatkan pengalaman pengguna dan ketahanan aplikasi. Namun, percobaan ulang yang sering dapat membebani bandwidth jaringan dan menyebabkan pertengkaran. Backoff eksponensial adalah teknik di mana operasi dicoba lagi dengan meningkatkan waktu tunggu untuk sejumlah upaya coba lagi tertentu.

Penerapan

Gunakan coba lagi dengan pola backoff saat:

- Layanan Anda sering membatasi permintaan untuk mencegah kelebihan beban, menghasilkan 429 Terlalu banyak permintaan pengecualian untuk proses panggilan.
- Jaringan adalah peserta yang tidak terlihat dalam arsitektur terdistribusi, dan masalah jaringan sementara mengakibatkan kegagalan.
- Layanan yang dipanggil sementara tidak tersedia, menyebabkan kegagalan. Percobaan ulang yang sering dapat menyebabkan degradasi layanan kecuali Anda memperkenalkan batas waktu backoff dengan menggunakan pola ini.

Masalah dan pertimbangan

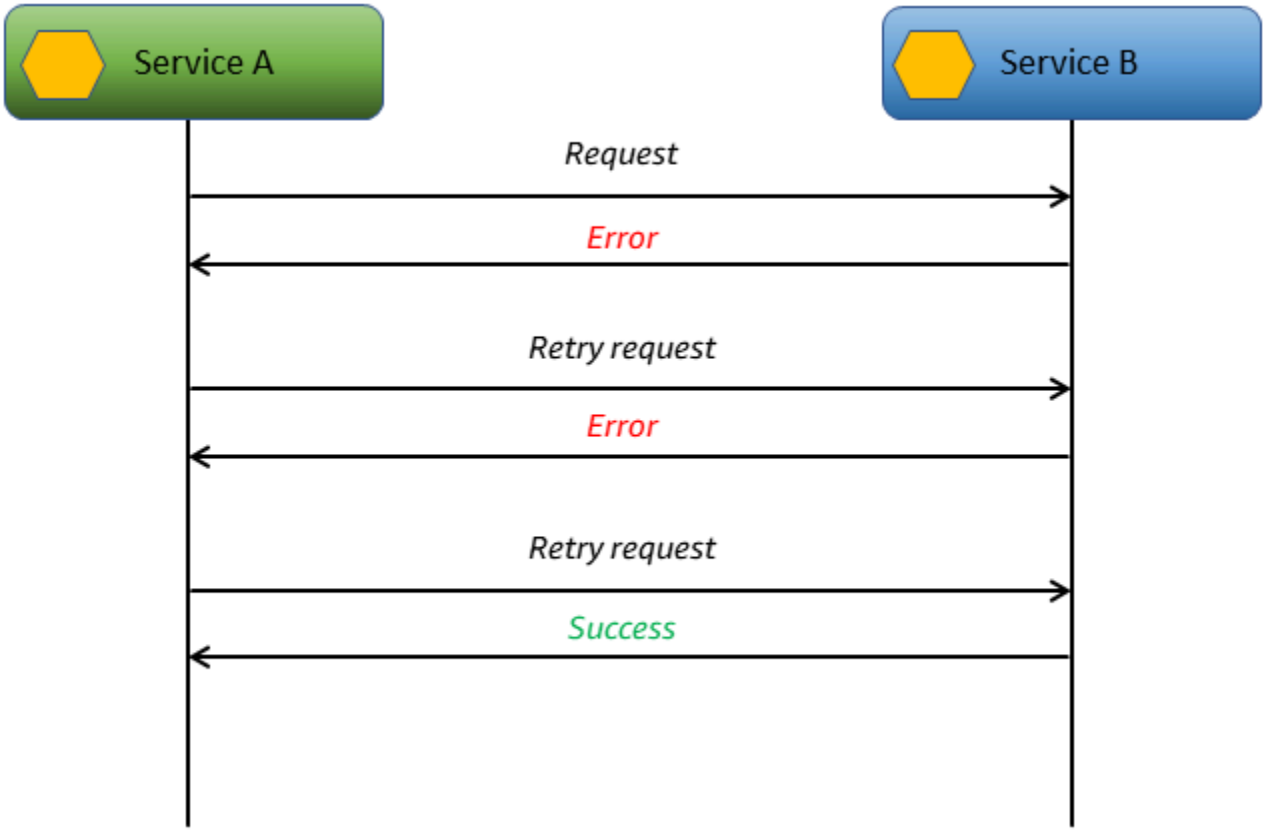
- Idempotensi: Jika beberapa panggilan ke metode memiliki efek yang sama dengan panggilan tunggal pada status sistem, operasi dianggap idempoten. Operasi harus idempoten saat Anda menggunakan coba lagi dengan pola backoff. Jika tidak, pembaruan sebagian dapat merusak status sistem.

- Bandwidth jaringan: Degradasi layanan dapat terjadi jika terlalu banyak percobaan ulang menempati bandwidth jaringan, yang menyebabkan waktu respons yang lambat.
- Gagal skenario cepat: Untuk kesalahan non-transien, jika Anda dapat menentukan penyebab kegagalan, akan lebih efisien untuk gagal cepat dengan menggunakan pola pemutus sirkuit.
- Tingkat backoff: Memperkenalkan backoff eksponensial dapat berdampak pada batas waktu layanan, menghasilkan waktu tunggu yang lebih lama bagi pengguna akhir.

Implementasi

Arsitektur tingkat tinggi

Diagram berikut menggambarkan bagaimana Layanan A dapat mencoba kembali panggilan ke Layanan B hingga respons yang berhasil dikembalikan. Jika Layanan B tidak mengembalikan respons yang berhasil setelah beberapa kali mencoba, Layanan A dapat berhenti mencoba lagi dan mengembalikan kegagalan ke pemanggilnya.

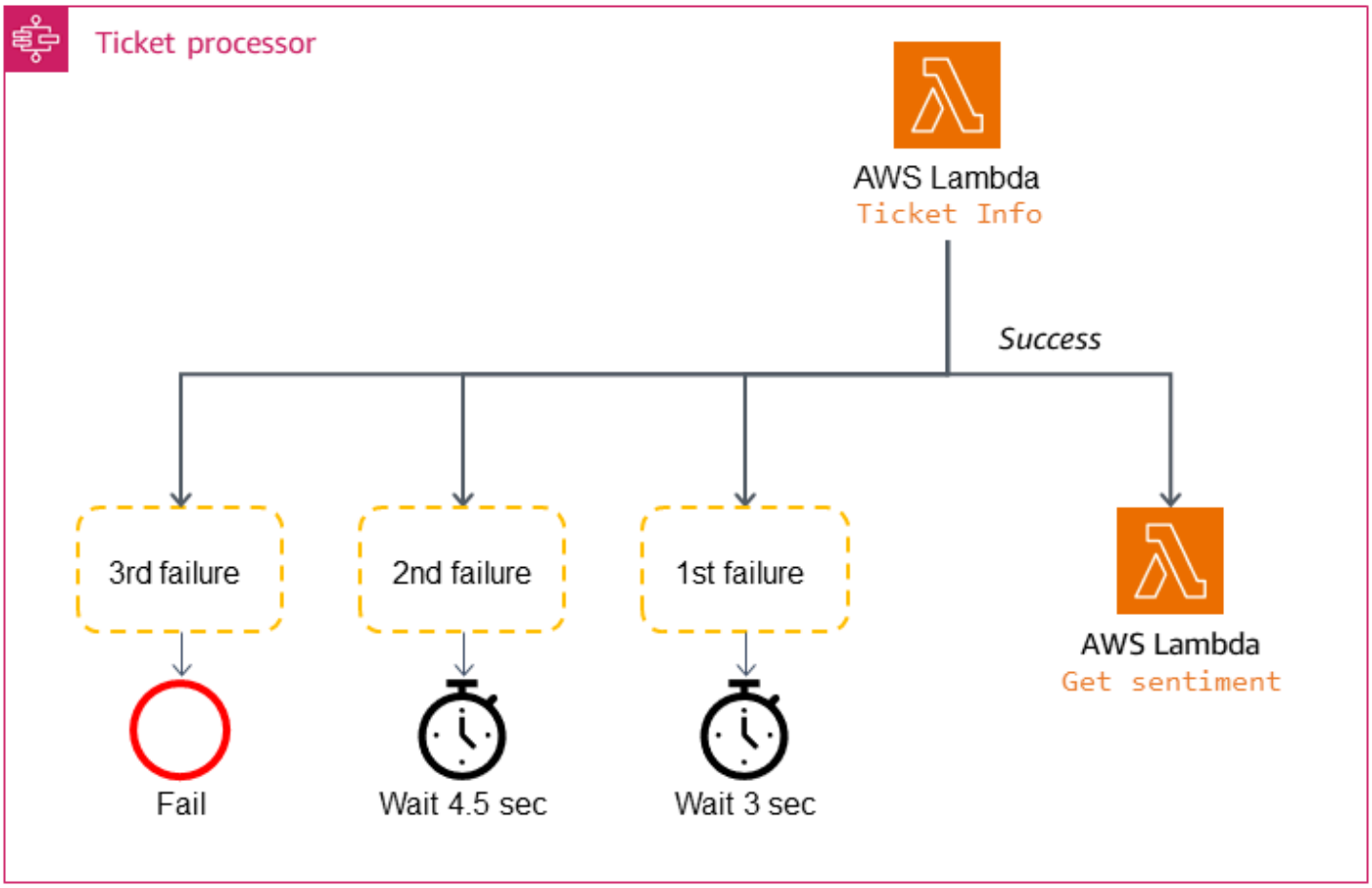


Implementasi menggunakan AWS layanan

Diagram berikut menunjukkan alur kerja pemrosesan tiket pada platform dukungan pelanggan. Tiket dari pelanggan yang tidak puas dipercepat dengan secara otomatis meningkatkan prioritas tiket. Fungsi `Ticket info` Lambda mengekstrak detail tiket dan memanggil fungsi Lambda`Get sentiment`. Fungsi `Get sentiment` Lambda memeriksa sentimen pelanggan dengan meneruskan deskripsi ke Amazon Comprehend (tidak ditampilkan).

Jika panggilan ke fungsi `Get sentiment` Lambda gagal, alur kerja mencoba ulang operasi tiga kali. AWS Step Functions memungkinkan backoff eksponensial dengan membiarkan Anda mengonfigurasi nilai backoff.

Dalam contoh ini, maksimal tiga percobaan ulang dikonfigurasi dengan pengganda peningkatan 1,5 detik. Jika percobaan ulang pertama terjadi setelah 3 detik, percobaan ulang kedua terjadi setelah 3 x 1,5 detik = 4,5 detik, dan percobaan ulang ketiga terjadi setelah 4,5 x 1,5 detik = 6,75 detik. Jika percobaan ulang ketiga tidak berhasil, alur kerja gagal. Logika backoff tidak memerlukan kode khusus apa pun–itu disediakan sebagai konfigurasi oleh. AWS Step Functions



Kode sampel

Kode berikut menunjukkan implementasi coba lagi dengan pola backoff.

```
public async Task DoRetriesWithBackOff()
  {
     int retries = 0;
     bool retry;
     do
     {
       //Sample object for sending parameters
       var parameterObj = new InputParameter { SimulateTimeout = "false" };
       var content = new StringContent(JsonConvert.SerializeObject(parameterObj),
                                 System.Text.Encoding.UTF8, "application/json");
       var waitInMilliseconds = Convert.ToInt32((Math.Pow(2, retries) - 1) * 100);
       System.Threading.Thread.Sleep(waitInMilliseconds);
       var response =  await _client.PostAsync(_baseURL, content);
       switch (response.StatusCode)
       {
         //Success
         case HttpStatusCode.OK:
           retry = false;
           Console.WriteLine(response.Content.ReadAsStringAsync().Result);
           break;
         //Throttling, timeouts
         case HttpStatusCode.TooManyRequests:
         case HttpStatusCode.GatewayTimeout:
           retry = true;
           break;
         //Some other error occured, so stop calling the API
         default:
           retry = false;
           break;
       }
       retries++;
     } while (retry && retries < MAX_RETRIES);
  }
```

GitHub repositori

Untuk implementasi lengkap arsitektur sampel untuk pola ini, lihat GitHub repositori di. https://github.com/aws-samples/retry-with-backoff

Konten terkait

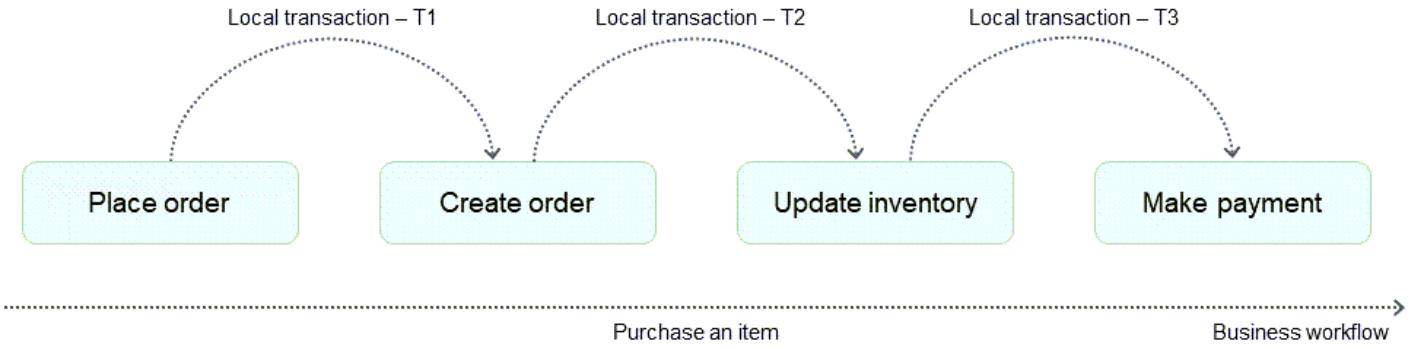
- Timeout, percobaan ulang, dan backoff dengan jitter (Amazon Builders' Library)

Pola saga

Sebuah saga terdiri dari urutan transaksi lokal. Setiap transaksi lokal dalam saga memperbarui database dan memicu transaksi lokal berikutnya. Jika transaksi gagal, saga menjalankan transaksi kompensasi untuk mengembalikan perubahan database yang dibuat oleh transaksi sebelumnya.

Urutan transaksi lokal ini membantu mencapai alur kerja bisnis dengan menggunakan prinsip kelanjutan dan kompensasi. Prinsip kelanjutan menentukan pemulihan alur kerja ke depan, sedangkan prinsip kompensasi menentukan pemulihan mundur. Jika pembaruan gagal pada setiap langkah dalam transaksi, saga menerbitkan acara untuk kelanjutan (untuk mencoba kembali transaksi) atau kompensasi (untuk kembali ke status data sebelumnya). Ini memastikan bahwa integritas data dipertahankan dan konsisten di seluruh penyimpanan data.

Misalnya, ketika pengguna membeli buku dari pengecer online, prosesnya terdiri dari urutan transaksi—seperti pembuatan pesanan, pembaruan inventaris, pembayaran, dan pengiriman—yang mewakili alur kerja bisnis. Untuk menyelesaikan alur kerja ini, arsitektur terdistribusi mengeluarkan urutan transaksi lokal untuk membuat pesanan dalam database pesanan, memperbarui database inventaris, dan memperbarui database pembayaran. Ketika proses berhasil, transaksi ini dipanggil secara berurutan untuk menyelesaikan alur kerja bisnis, seperti yang ditunjukkan diagram berikut. Namun, jika salah satu dari transaksi lokal ini gagal, sistem harus dapat memutuskan langkah berikutnya yang tepat — yaitu, pemulihan ke depan atau pemulihan mundur.



Dua skenario berikut membantu menentukan apakah langkah selanjutnya adalah pemulihan maju atau pemulihan mundur:

- Kegagalan tingkat platform, di mana ada yang tidak beres dengan infrastruktur yang mendasarinya dan menyebabkan transaksi gagal. Dalam hal ini, pola saga dapat melakukan pemulihan ke depan dengan mencoba kembali transaksi lokal dan melanjutkan proses bisnis.

- Kegagalan tingkat aplikasi, di mana layanan pembayaran gagal karena pembayaran yang tidak valid. Dalam hal ini, pola saga dapat melakukan pemulihan mundur dengan mengeluarkan transaksi kompensasi untuk memperbarui inventaris dan database pesanan, dan mengembalikan keadaan sebelumnya.

Pola saga menangani alur kerja bisnis dan memastikan bahwa keadaan akhir yang diinginkan dicapai melalui pemulihan ke depan. Jika terjadi kegagalan, itu mengembalikan transaksi lokal dengan menggunakan pemulihan mundur untuk menghindari masalah konsistensi data.

Pola saga memiliki dua varian: koreografi dan orkestrasi.

Koreografi Saga

Pola koreografi saga tergantung pada peristiwa yang diterbitkan oleh layanan mikro. Peserta saga (layanan mikro) berlangganan acara dan bertindak berdasarkan pemicu acara. Misalnya, layanan pesanan dalam diagram berikut memancarkan suatu `OrderPlaced` peristiwa. Layanan inventaris berlangganan acara itu dan memperbarui inventaris saat `OrderPlaced` acara dipancarkan. Demikian pula, layanan peserta bertindak berdasarkan konteks peristiwa yang dipancarkan.

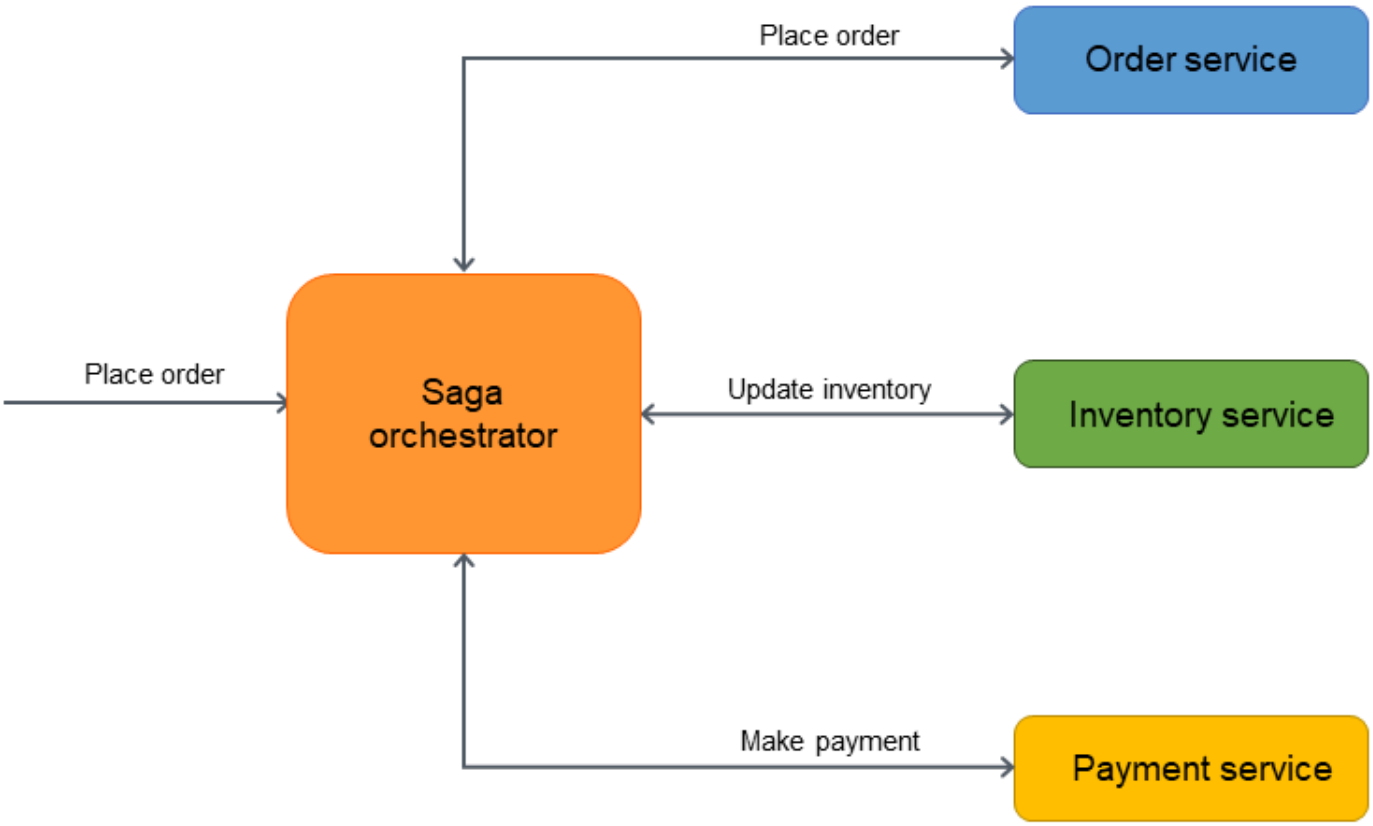
Pola koreografi saga cocok ketika hanya ada beberapa peserta dalam saga, dan Anda memerlukan implementasi sederhana tanpa satu titik kegagalan. Ketika lebih banyak peserta ditambahkan, menjadi lebih sulit untuk melacak dependensi antara peserta dengan menggunakan pola ini.

Untuk ulasan terperinci, lihat bagian koreografi Saga dari panduan ini.

Orkestrasi Saga

Pola orkestrasi saga memiliki koordinator pusat yang disebut orkestrator. Orkestrator saga mengelola dan mengoordinasikan seluruh siklus hidup transaksi. Ia menyadari serangkaian langkah yang harus dilakukan untuk menyelesaikan transaksi. Untuk menjalankan langkah, ia mengirim pesan ke layanan mikro peserta untuk melakukan operasi. Layanan mikro peserta menyelesaikan operasi dan mengirim pesan kembali ke orkestrator. Berdasarkan pesan yang diterimanya, orkestrator memutuskan layanan mikro mana yang akan dijalankan selanjutnya dalam transaksi.

Pola orkestrasi saga cocok ketika ada banyak peserta, dan kopling longgar diperlukan antara peserta saga. Orkestrator merangkum kompleksitas dalam logika dengan membuat para peserta digabungkan secara longgar. Namun, orkestrator dapat menjadi satu titik kegagalan karena mengontrol seluruh alur kerja.



Untuk ulasan terperinci, lihat bagian orkestrasi Saga dari panduan ini.

Pola koreografi Saga

Niat

Pola koreografi saga membantu menjaga integritas data dalam transaksi terdistribusi yang menjangkau beberapa layanan dengan menggunakan langganan acara. Dalam transaksi terdistribusi, beberapa layanan dapat dipanggil sebelum transaksi selesai. Ketika layanan menyimpan data di penyimpanan data yang berbeda, mungkin sulit untuk mempertahankan konsistensi data di seluruh penyimpanan data ini.

Motivasi

Transaksi adalah satu unit kerja yang mungkin melibatkan beberapa langkah, di mana semua langkah dijalankan sepenuhnya atau tidak ada langkah yang dijalankan, menghasilkan penyimpanan data yang mempertahankan status konsistennya. Istilah atomisitas, konsistensi, isolasi, dan daya tahan (ACID) mendefinisikan sifat transaksi. Database relasional menyediakan transaksi ACID untuk menjaga konsistensi data.

Untuk menjaga konsistensi dalam suatu transaksi, database relasional menggunakan metode two-phase commit (2PC). Ini terdiri dari fase persiapan dan fase komit.

- Pada tahap persiapan, proses koordinasi meminta proses partisipasi transaksi (peserta) untuk berjanji untuk melakukan atau memutar kembali transaksi.
- Pada fase komit, proses koordinasi meminta peserta untuk melakukan transaksi. Jika peserta tidak setuju untuk berkomitmen dalam fase persiapan, transaksi dibatalkan.

Dalam sistem terdistribusi yang mengikuti pola database-per-service desain, komit dua fase bukanlah pilihan. Ini karena setiap transaksi didistribusikan di berbagai database, dan tidak ada pengontrol tunggal yang dapat mengoordinasikan proses yang mirip dengan komit dua fase di penyimpanan data relasional. Dalam hal ini, salah satu solusinya adalah dengan menggunakan pola koreografi saga.

Penerapan

Gunakan pola koreografi saga saat:

- Sistem Anda membutuhkan integritas dan konsistensi data dalam transaksi terdistribusi yang mencakup beberapa penyimpanan data.
- Penyimpanan data (misalnya, database NoSQL) tidak menyediakan 2PC untuk menyediakan transaksi ACID, Anda perlu memperbarui beberapa tabel dalam satu transaksi, dan menerapkan 2PC dalam batas aplikasi akan menjadi tugas yang kompleks.
- Proses pengendalian pusat yang mengelola transaksi peserta dapat menjadi satu titik kegagalan.
- Peserta saga adalah layanan independen dan perlu digabungkan secara longgar.
- Ada komunikasi antara konteks terbatas dalam domain bisnis.

Masalah dan pertimbangan

- Kompleksitas: Ketika jumlah layanan mikro meningkat, koreografi saga dapat menjadi sulit untuk dikelola karena jumlah interaksi antara layanan mikro. Selain itu, transaksi kompensasi dan percobaan ulang menambah kompleksitas pada kode aplikasi, yang dapat mengakibatkan overhead pemeliharaan. Koreografi cocok ketika hanya ada beberapa peserta dalam saga, dan Anda memerlukan implementasi sederhana tanpa satu titik kegagalan. Ketika lebih banyak peserta ditambahkan, menjadi lebih sulit untuk melacak dependensi antara peserta dengan menggunakan pola ini.
- Implementasi tangguh: Dalam koreografi saga, lebih sulit untuk menerapkan batas waktu, percobaan ulang, dan pola ketahanan lainnya secara global, dibandingkan dengan orkestrasi saga. Koreografi harus diimplementasikan pada komponen individu, bukan pada tingkat orkestrator.
- Dependensi siklik: Para peserta mengkonsumsi pesan yang diterbitkan oleh satu sama lain. Hal ini dapat mengakibatkan dependensi siklik, yang mengarah ke kompleksitas kode dan overhead pemeliharaan, dan kemungkinan kebuntuan.
- Masalah penulisan ganda: Layanan mikro harus memperbarui database secara atomik dan mempublikasikan suatu peristiwa. Kegagalan kedua operasi dapat menyebabkan keadaan yang tidak konsisten. Salah satu cara untuk mengatasi ini adalah dengan menggunakan pola outbox transaksional.
- Melestarikan acara: Para peserta saga bertindak berdasarkan peristiwa yang diterbitkan. Sangat penting untuk menyimpan peristiwa dalam urutan yang terjadi untuk tujuan audit, debugging, dan replay. Anda dapat menggunakan pola sumber peristiwa untuk mempertahankan peristiwa di penyimpanan acara jika pemutaran ulang status sistem diperlukan untuk memulihkan konsistensi data. Toko acara juga dapat digunakan untuk tujuan audit dan pemecahan masalah karena mencerminkan setiap perubahan dalam sistem.
- Konsistensi akhir: Pemrosesan berurutan transaksi lokal menghasilkan konsistensi akhirnya, yang dapat menjadi tantangan dalam sistem yang membutuhkan konsistensi yang kuat. Anda dapat mengatasi masalah ini dengan menetapkan harapan tim bisnis Anda untuk model konsistensi atau menilai kembali kasus penggunaan dan beralih ke database yang memberikan konsistensi yang kuat.
- Idempotensi: Peserta Saga harus idempoten untuk memungkinkan eksekusi berulang jika terjadi kegagalan sementara yang disebabkan oleh crash tak terduga dan kegagalan orkestrator.
- Isolasi transaksi: Pola saga tidak memiliki isolasi transaksi, yang merupakan salah satu dari empat properti dalam transaksi ACID. Tingkat isolasi transaksi menentukan seberapa banyak transaksi bersamaan lainnya dapat mempengaruhi data tempat transaksi beroperasi. Orkestrasi transaksi

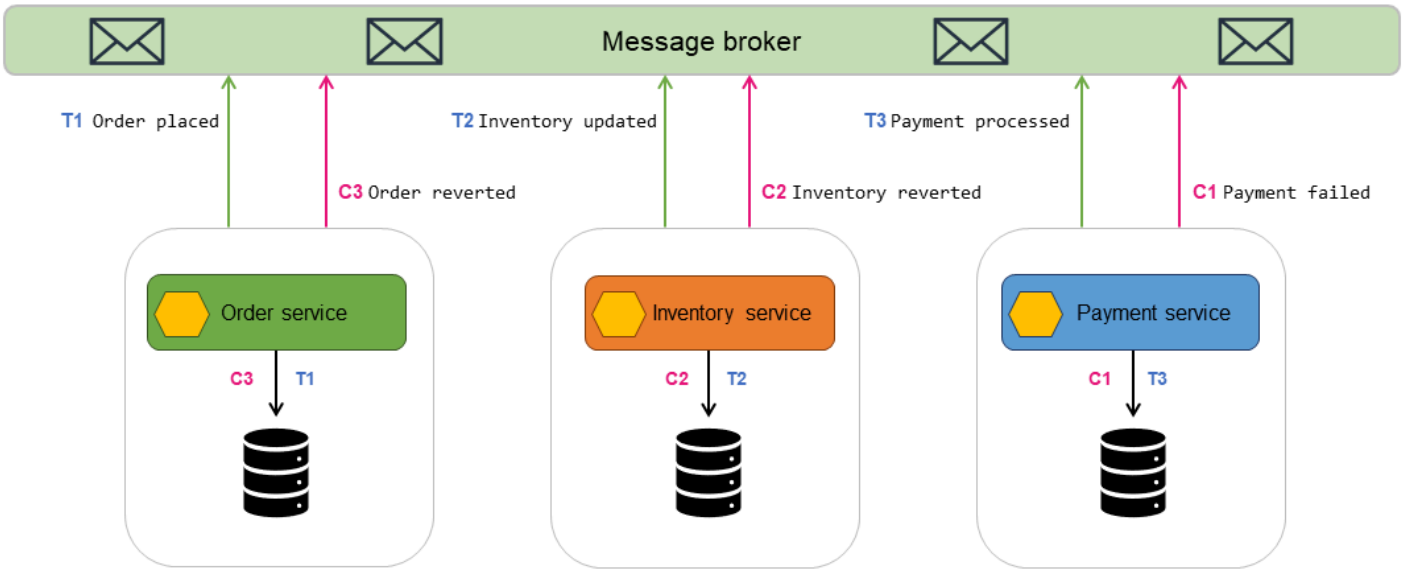
secara bersamaan dapat menyebabkan data basi. Kami merekomendasikan menggunakan penguncian semantik untuk menangani skenario seperti itu.

- Observabilitas: Observabilitas mengacu pada pencatatan dan penelusuran terperinci untuk memecahkan masalah dalam proses implementasi dan orkestrasi. Ini menjadi penting ketika jumlah peserta saga meningkat, menghasilkan kompleksitas dalam debugging. End-to-endpemantauan dan pelaporan lebih sulit dicapai dalam koreografi saga, dibandingkan dengan orkestrasi saga.
- Masalah latensi: Transaksi kompensasi dapat menambah latensi ke waktu respons keseluruhan ketika saga terdiri dari beberapa langkah. Jika transaksi melakukan panggilan sinkron, ini dapat meningkatkan latensi lebih lanjut.

Implementasi

Arsitektur tingkat tinggi

Dalam diagram arsitektur berikut, koreografi saga memiliki tiga peserta: layanan pemesanan, layanan inventaris, dan layanan pembayaran. Tiga langkah diperlukan untuk menyelesaikan transaksi: T1, T2, dan T3. Tiga transaksi kompensasi mengembalikan data ke keadaan awal: C1, C2, dan C3.



- Layanan pesanan menjalankan transaksi lokal, T1, yang secara atomik memperbarui database dan menerbitkan `Order placed` pesan ke broker pesan.
- Layanan inventaris berlangganan pesan layanan pesanan dan menerima pesan bahwa pesanan telah dibuat.

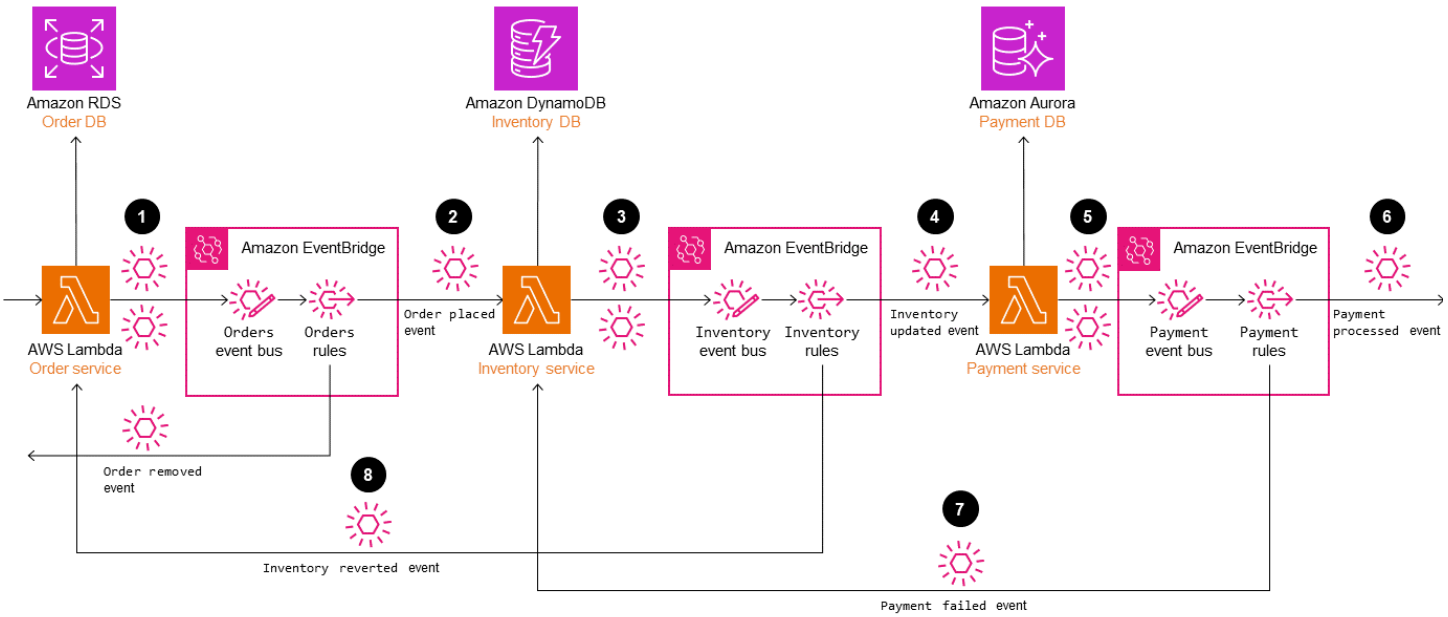
- Layanan inventaris menjalankan transaksi lokal, T2, yang secara atomik memperbarui database dan menerbitkan `Inventory updated` pesan ke broker pesan.
- Layanan pembayaran berlangganan pesan dari layanan inventaris dan menerima pesan bahwa inventaris telah diperbarui.
- Layanan pembayaran menjalankan transaksi lokal, T3, yang secara atomik memperbarui database dengan detail pembayaran dan menerbitkan pesan ke `Payment processed` broker pesan.
- Jika pembayaran gagal, layanan pembayaran menjalankan transaksi kompensasi, C1, yang secara atomik mengembalikan pembayaran dalam database dan menerbitkan pesan ke broker pesan. `Payment failed`
- Transaksi kompensasi C2 dan C3 dijalankan untuk mengembalikan konsistensi data.

Implementasi menggunakan layanan AWS

Anda dapat menerapkan pola koreografi saga dengan menggunakan Amazon. EventBridge EventBridgemenggunakan acara untuk menghubungkan komponen aplikasi. Ini memproses peristiwa melalui bus acara atau pipa. Bus acara adalah router yang menerima acara dan mengirimkannya ke nol atau lebih tujuan, atau target. Aturan yang terkait dengan bus acara mengevaluasi peristiwa saat mereka tiba dan mengirimkannya ke target untuk diproses.

Dalam arsitektur berikut:

- Layanan mikro — layanan pemesanan, layanan inventaris, dan layanan pembayaran — diimplementasikan sebagai fungsi Lambda.
- Ada tiga EventBridge bus khusus: bus `Orders` acara, bus `Inventory` acara, dan bus `Payment` acara.
- `Orders`saturan, `Inventory` aturan, dan `Payment` aturan cocok dengan peristiwa yang dikirim ke bus acara terkait dan memanggil fungsi Lambda.



Dalam skenario yang sukses, ketika pesanan ditempatkan:

1. Layanan pesanan memproses permintaan dan mengirimkan acara ke bus `Orders` acara.
2. `Orders`Aturan cocok dengan acara dan memulai layanan inventaris.
3. Layanan inventaris memperbarui inventaris dan mengirimkan acara ke bus `Inventory` acara.
4. `Inventory`Aturan cocok dengan acara dan memulai layanan pembayaran.
5. Layanan pembayaran memproses pembayaran dan mengirimkan acara ke bus `Payment` acara.
6. `Payment`Aturan cocok dengan acara dan mengirim pemberitahuan `Payment processed` acara ke pendengar.

   Atau, ketika ada masalah dalam pemrosesan pesanan, EventBridge aturan memulai transaksi kompensasi untuk mengembalikan pembaruan data untuk menjaga konsistensi dan integritas data.
7. Jika pembayaran gagal, `Payment` aturan memproses acara dan memulai layanan inventaris.Layanan inventaris menjalankan transaksi kompensasi untuk mengembalikan inventaris.
8. Ketika inventaris telah dikembalikan, layanan inventaris mengirimkan *`Inventory reverted`* acara ke bus *`Inventory`* acara.Acara ini diproses oleh `Inventory` aturan.Ini memulai layanan pesanan, yang menjalankan transaksi kompensasi untuk menghapus pesanan.

Konten terkait

- Pola orkestrasi Saga
- Pola kotak keluar transaksional
- Coba lagi dengan pola backoff

Pola orkestrasi Saga

Niat

Pola orkestrasi saga menggunakan koordinator pusat (orkestrator) untuk membantu menjaga integritas data dalam transaksi terdistribusi yang menjangkau beberapa layanan. Dalam transaksi terdistribusi, beberapa layanan dapat dipanggil sebelum transaksi selesai. Ketika layanan menyimpan data di penyimpanan data yang berbeda, mungkin sulit untuk mempertahankan konsistensi data di seluruh penyimpanan data ini.

Motivasi

Transaksi adalah satu unit kerja yang mungkin melibatkan beberapa langkah, di mana semua langkah dijalankan sepenuhnya atau tidak ada langkah yang dijalankan, menghasilkan penyimpanan data yang mempertahankan status konsistennya. Istilah atomisitas, konsistensi, isolasi, dan daya tahan (ACID) mendefinisikan sifat transaksi. Database relasional menyediakan transaksi ACID untuk menjaga konsistensi data.

Untuk menjaga konsistensi dalam suatu transaksi, database relasional menggunakan metode two-phase commit (2PC). Ini terdiri dari fase persiapan dan fase komit.

- Pada tahap persiapan, proses koordinasi meminta proses partisipasi transaksi (peserta) untuk berjanji untuk melakukan atau memutar kembali transaksi.
- Pada fase komit, proses koordinasi meminta peserta untuk melakukan transaksi. Jika peserta tidak setuju untuk berkomitmen dalam fase persiapan, transaksi dibatalkan.

Dalam sistem terdistribusi yang mengikuti pola database-per-service desain, komit dua fase bukanlah pilihan. Ini karena setiap transaksi didistribusikan di berbagai database, dan tidak ada pengontrol tunggal yang dapat mengoordinasikan proses yang mirip dengan komit dua fase di penyimpanan data relasional. Dalam hal ini, salah satu solusinya adalah dengan menggunakan pola orkestrasi saga.

Penerapan

Gunakan pola orkestrasi saga saat:

- Sistem Anda membutuhkan integritas dan konsistensi data dalam transaksi terdistribusi yang mencakup beberapa penyimpanan data.
- Penyimpanan data tidak menyediakan 2PC untuk menyediakan transaksi ACID, dan menerapkan 2PC dalam batas aplikasi adalah tugas yang kompleks.
- Anda memiliki database NoSQL, yang tidak menyediakan transaksi ACID, dan Anda perlu memperbarui beberapa tabel dalam satu transaksi.

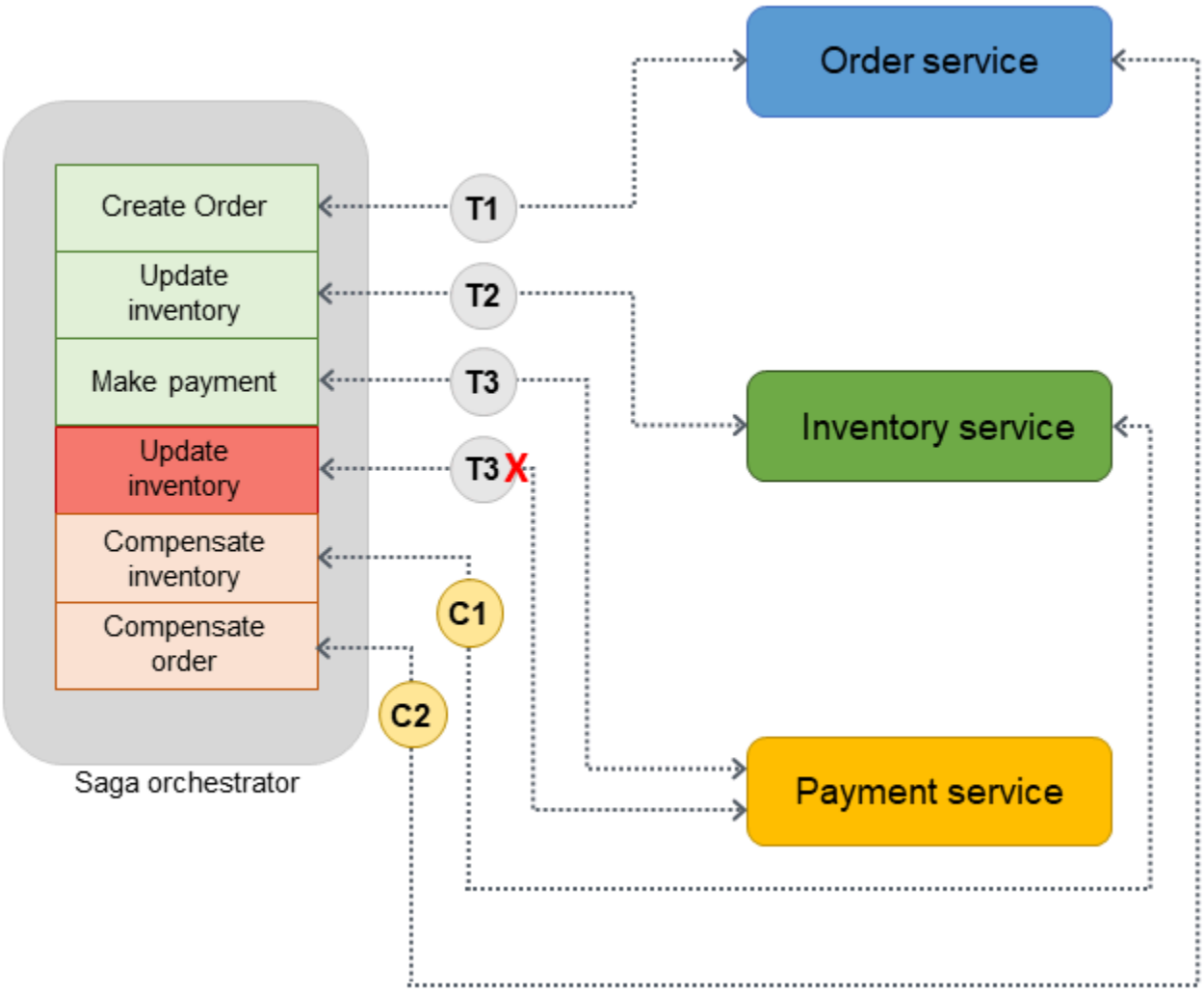
Masalah dan pertimbangan

- Kompleksitas: Transaksi kompensasi dan percobaan ulang menambah kompleksitas pada kode aplikasi, yang dapat mengakibatkan overhead pemeliharaan.
- Konsistensi akhir: Pemrosesan berurutan transaksi lokal menghasilkan konsistensi akhirnya, yang dapat menjadi tantangan dalam sistem yang membutuhkan konsistensi yang kuat. Anda dapat mengatasi masalah ini dengan menetapkan harapan tim bisnis Anda untuk model konsistensi atau dengan beralih ke penyimpanan data yang memberikan konsistensi yang kuat.
- Idempotensi: Peserta Saga harus idempoten untuk memungkinkan eksekusi berulang jika terjadi kegagalan sementara yang disebabkan oleh crash tak terduga dan kegagalan orkestrator.
- Isolasi transaksi: Saga tidak memiliki isolasi transaksi. Orkestrasi transaksi secara bersamaan dapat menyebabkan data basi. Kami merekomendasikan menggunakan penguncian semantik untuk menangani skenario seperti itu.
- Observabilitas: Observabilitas mengacu pada pencatatan dan penelusuran terperinci untuk memecahkan masalah dalam proses eksekusi dan orkestrasi. Ini menjadi penting ketika jumlah peserta saga meningkat, menghasilkan kompleksitas dalam debugging.
- Masalah latensi: Transaksi kompensasi dapat menambah latensi ke waktu respons keseluruhan ketika saga terdiri dari beberapa langkah. Hindari panggilan sinkron dalam kasus seperti itu.
- Titik kegagalan tunggal: Orkestrator dapat menjadi satu titik kegagalan karena mengoordinasikan seluruh transaksi. Dalam beberapa kasus, pola koreografi saga lebih disukai karena masalah ini.

Implementasi

Arsitektur tingkat tinggi

Dalam diagram arsitektur berikut, orkestrator saga memiliki tiga peserta: layanan pesanan, layanan inventaris, dan layanan pembayaran. Tiga langkah diperlukan untuk menyelesaikan transaksi: T1, T2, dan T3. Orkestrator saga menyadari langkah-langkah dan menjalankannya dalam urutan yang diperlukan. Ketika langkah T3 gagal (kegagalan pembayaran), orkestrator menjalankan transaksi kompensasi C1 dan C2 untuk mengembalikan data ke keadaan awal.



Anda dapat menggunakan AWS Step Functionsuntuk menerapkan orkestrasi saga ketika transaksi didistribusikan di beberapa database.

Implementasi menggunakan AWS layanan

Solusi sampel menggunakan alur kerja standar di Step Functions untuk mengimplementasikan pola orkestrasi saga.

Saat pelanggan memanggil API, fungsi Lambda dipanggil, dan preprocessing terjadi di fungsi Lambda. Fungsi ini memulai alur kerja Step Functions untuk mulai memproses transaksi terdistribusi. Jika preprocessing tidak diperlukan, Anda dapat memulai alur kerja Step Functions langsung dari API Gateway tanpa menggunakan fungsi Lambda.

Penggunaan Step Functions mengurangi titik tunggal masalah kegagalan, yang melekat dalam implementasi pola orkestrasi saga. Step Functions memiliki toleransi kesalahan bawaan dan mempertahankan kapasitas layanan di beberapa Availability Zone di setiap Wilayah AWS untuk melindungi aplikasi dari kegagalan mesin atau pusat data individual. Ini membantu memastikan ketersediaan yang tinggi untuk layanan itu sendiri dan untuk alur kerja aplikasi yang dioperasikannya.

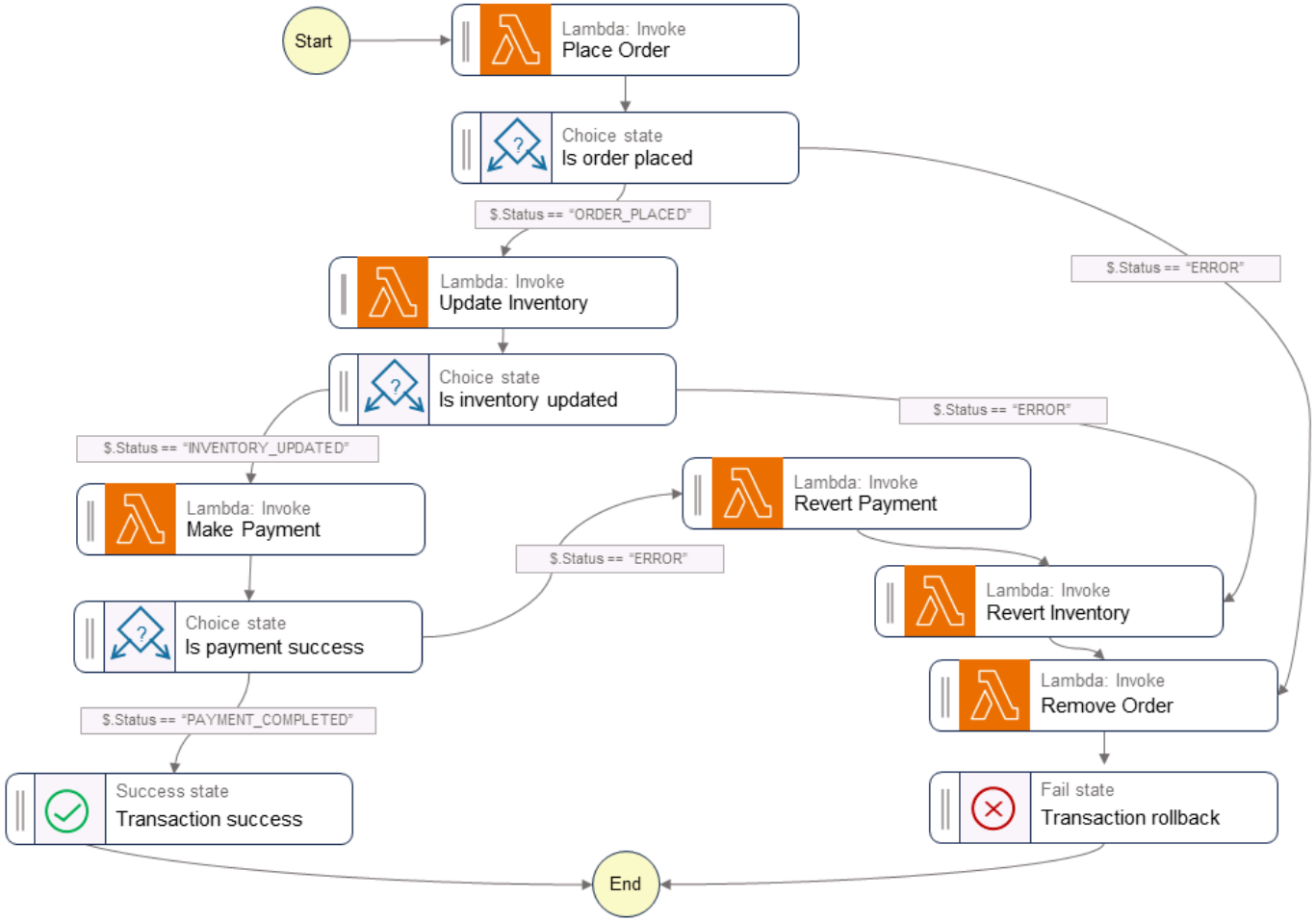
Alur kerja Step Functions

Mesin status Step Functions memungkinkan Anda mengonfigurasi persyaratan alur kontrol berbasis keputusan untuk implementasi pola. Alur kerja Step Functions memanggil layanan individual untuk penempatan pesanan, pembaruan inventaris, dan pemrosesan pembayaran untuk menyelesaikan transaksi dan mengirimkan pemberitahuan peristiwa untuk diproses lebih lanjut. Alur kerja Step Functions bertindak sebagai orkestrator untuk mengoordinasikan transaksi. Jika alur kerja berisi kesalahan, orkestrator menjalankan transaksi kompensasi untuk memastikan integritas data dipertahankan di seluruh layanan.

Diagram berikut menunjukkan langkah-langkah yang berjalan di dalam alur kerja Step Functions. Langkah `Place Order`, `Update Inventory`, dan `Make Payment` langkah-langkah menunjukkan jalan sukses. Pesanan ditempatkan, inventaris diperbarui, dan pembayaran diproses sebelum `Success` status dikembalikan ke penelepon.

Fungsi`Revert Payment`,`Revert Inventory`, dan `Remove Order` Lambda menunjukkan transaksi kompensasi yang dijalankan orkestrator ketika setiap langkah dalam alur kerja gagal. Jika alur kerja gagal pada `Update Inventory` langkah tersebut, orkestrator memanggil `Revert`

`Inventory` dan `Remove Order` langkah-langkah sebelum mengembalikan `Fail` status ke pemanggil. Transaksi kompensasi ini memastikan bahwa integritas data dipertahankan. Inventaris kembali ke tingkat semula dan pesanan dikembalikan.



Kode sampel

Contoh kode berikut menunjukkan bagaimana Anda dapat membuat orkestrator saga dengan menggunakan Step Functions. Untuk melihat kode lengkap, lihat GitHubrepositori untuk contoh ini.

Ketentuan tugas

```
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);
```

Fungsi langkah dan definisi mesin negara

```
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 repositori

Untuk implementasi lengkap arsitektur sampel untuk pola ini, lihat GitHub repositori di. https://github.com/aws-samples/saga-orchestration-netcore-blog

Referensi blog

- Membangun aplikasi terdistribusi tanpa server menggunakan pola Orkestrasi Saga

Konten terkait

- Pola koreografi Saga
- Pola kotak keluar transaksional

Video

Video berikut membahas bagaimana menerapkan pola orkestrasi saga dengan menggunakan. AWS Step Functions

Pola sebar-kumpulkan

Niat

Pola scatter-gathering adalah pola perutean pesan yang melibatkan penyiaran permintaan serupa atau terkait ke beberapa penerima, dan menggabungkan tanggapan mereka kembali ke dalam satu pesan dengan menggunakan komponen yang disebut agregator. Pola ini membantu mencapai paralelisasi, mengurangi latensi pemrosesan, dan menangani komunikasi asinkron. Sangat mudah untuk menerapkan pola scatter-gathering dengan menggunakan pendekatan sinkron, tetapi pendekatan yang lebih kuat melibatkan penerapannya sebagai perutean pesan dalam komunikasi asinkron, baik dengan atau tanpa layanan pesan.

Motivasi

Dalam pemrosesan aplikasi, permintaan yang mungkin membutuhkan waktu lama untuk diproses secara berurutan dapat dibagi menjadi beberapa permintaan yang diproses secara paralel. Anda juga dapat mengirim permintaan ke beberapa sistem eksternal melalui panggilan API untuk mendapatkan respons. Pola scatter-gathering berguna ketika Anda membutuhkan masukan dari berbagai sumber. Scatter-gathering mengumpulkan hasil untuk membantu Anda membuat keputusan berdasarkan informasi atau untuk memilih respons terbaik untuk permintaan tersebut.

Pola scatter-gathering terdiri dari dua fase, seperti namanya:

- Fase pencar memproses pesan permintaan dan mengirimkannya ke beberapa penerima secara paralel. Selama fase ini, aplikasi menyebarkan permintaan di seluruh jaringan dan terus berjalan tanpa menunggu tanggapan segera.
- Selama fase pengumpulan, aplikasi mengumpulkan tanggapan dari penerima, dan menyaring atau menggabungkannya menjadi respons terpadu. Ketika semua tanggapan telah dikumpulkan, mereka dapat digabungkan menjadi satu respons atau yang terbaik dapat dipilih untuk diproses lebih lanjut.

Penerapan

Gunakan pola scatter-gathering saat:

- Anda berencana untuk mengumpulkan dan mengkonsolidasikan data dari berbagai APIs untuk membuat respons yang akurat. Pola tersebut mengkonsolidasikan informasi dari sumber yang berbeda menjadi keseluruhan yang kohesif. Misalnya, sistem pemesanan dapat mengajukan permintaan ke beberapa penerima untuk mendapatkan penawaran dari beberapa mitra eksternal.
- Permintaan yang sama harus dikirim ke beberapa penerima secara bersamaan untuk menyelesaikan transaksi. Misalnya, Anda dapat menggunakan pola ini untuk menanyakan data inventaris secara paralel untuk memeriksa ketersediaan produk.
- Anda ingin menerapkan sistem yang andal dan terukur di mana load balancing dapat dicapai dengan mendistribusikan permintaan ke beberapa penerima. Jika satu penerima gagal atau mengalami beban tinggi, penerima lain masih dapat memproses permintaan.
- Anda ingin mengoptimalkan kinerja saat menerapkan kueri kompleks yang melibatkan beberapa sumber data. Anda dapat menyebarkan kueri ke database yang relevan, mengumpulkan sebagian hasil, dan menggabungkannya menjadi jawaban yang komprehensif.
- Anda menerapkan jenis pemrosesan pengurangan peta di mana permintaan data dirutekan ke beberapa titik akhir pemrosesan data untuk sharding dan replikasi. Hasil sebagian disaring dan digabungkan untuk menyusun respons yang tepat.
- Anda ingin mendistribusikan operasi tulis di ruang kunci partisi dalam beban kerja berat tulis di database nilai kunci. Agregator membaca hasilnya dengan menanyakan data di setiap pecahan, dan kemudian mengkonsolidasikannya menjadi satu respons.

Masalah dan pertimbangan

- Toleransi kesalahan: Pola ini bergantung pada beberapa penerima yang bekerja secara paralel, jadi penting untuk menangani kegagalan dengan anggun. Untuk mengurangi dampak kegagalan penerima pada keseluruhan sistem, Anda dapat menerapkan strategi seperti redundansi, replikasi, dan deteksi kesalahan.
- Batas penskalaan: Ketika jumlah total node pemrosesan meningkat, overhead jaringan terkait juga meningkat. Setiap permintaan yang melibatkan komunikasi melalui jaringan dapat meningkatkan latensi dan berdampak negatif pada manfaat paralelisasi.
- Kemacetan waktu respons: Untuk operasi yang mengharuskan semua penerima diproses sebelum pemrosesan akhir dilakukan, kinerja sistem secara keseluruhan dibatasi oleh waktu respons penerima yang paling lambat.
- Respons sebagian: Ketika permintaan tersebar ke beberapa penerima, beberapa penerima dapat time out. Dalam kasus ini, implementasi harus berkomunikasi dengan klien bahwa responsnya

tidak lengkap. Anda juga dapat menampilkan detail agregasi respons dengan menggunakan frontend UI.

- Konsistensi data: Saat Anda memproses data di beberapa penerima, Anda harus mempertimbangkan dengan cermat teknik sinkronisasi data dan resolusi konflik, untuk memastikan bahwa hasil agregat akhir akurat dan konsisten.

Implementasi

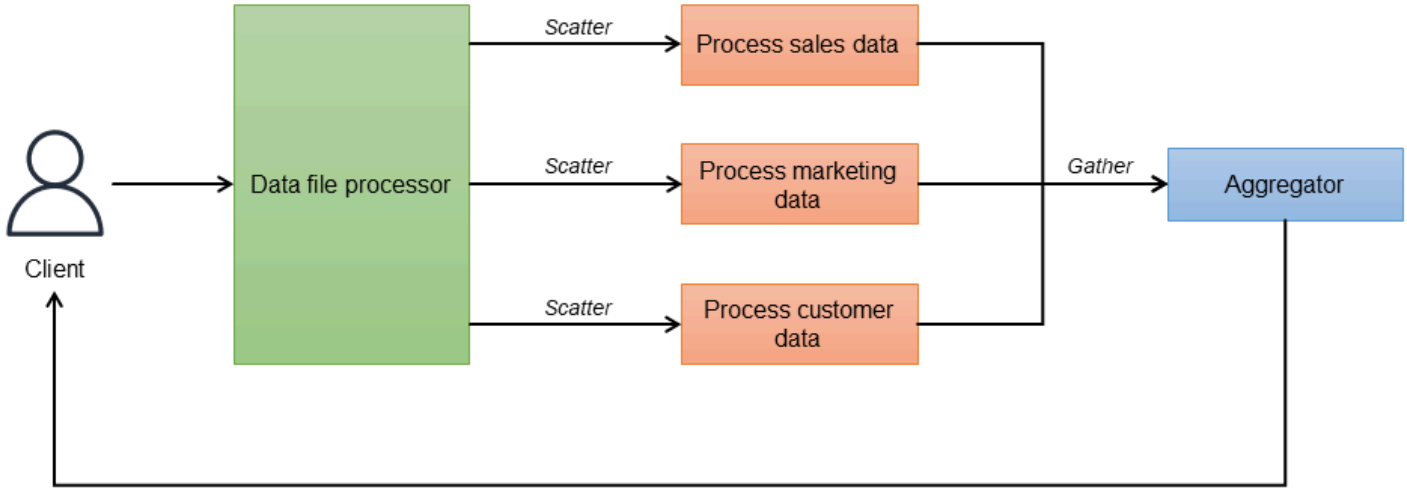
Arsitektur tingkat tinggi

Pola scatter-gathering menggunakan root controller untuk mendistribusikan permintaan ke penerima yang akan memproses permintaan. Selama fase pencar, pola ini dapat menggunakan dua mekanisme untuk mengirim pesan ke penerima:

- Scatter by distribution: Aplikasi ini memiliki daftar penerima yang diketahui yang harus dipanggil untuk mendapatkan hasilnya. Penerima dapat berupa proses berbeda yang memiliki fungsi unik atau satu proses yang telah diskalakan untuk mendistribusikan beban pemrosesan. Jika salah satu node pemrosesan habis waktu atau menunjukkan keterlambatan dalam merespons, pengontrol dapat mendistribusikan kembali pemrosesan ke node lain.
- Scatter by auction: Aplikasi menyiarkan pesan ke penerima yang tertarik dengan menggunakan pola berlangganan publikasi. Dalam hal ini, penerima dapat berlangganan pesan atau menarik diri dari langganan kapan saja.

Sebarkan dengan distribusi

Dalam metode pencar dengan distribusi, pengontrol root membagi permintaan yang masuk menjadi tugas independen dan menetapkannya ke penerima yang tersedia (fase pencar). Setiap penerima (proses, wadah, atau fungsi Lambda) bekerja secara independen dan paralel pada perhitungannya, dan menghasilkan sebagian dari respons. Ketika penerima menyelesaikan tugas mereka, mereka mengirim tanggapan mereka ke agregator (fase pengumpulan). Agregator menggabungkan sebagian tanggapan dan mengembalikan hasil akhir ke klien. Diagram berikut menggambarkan alur kerja ini.



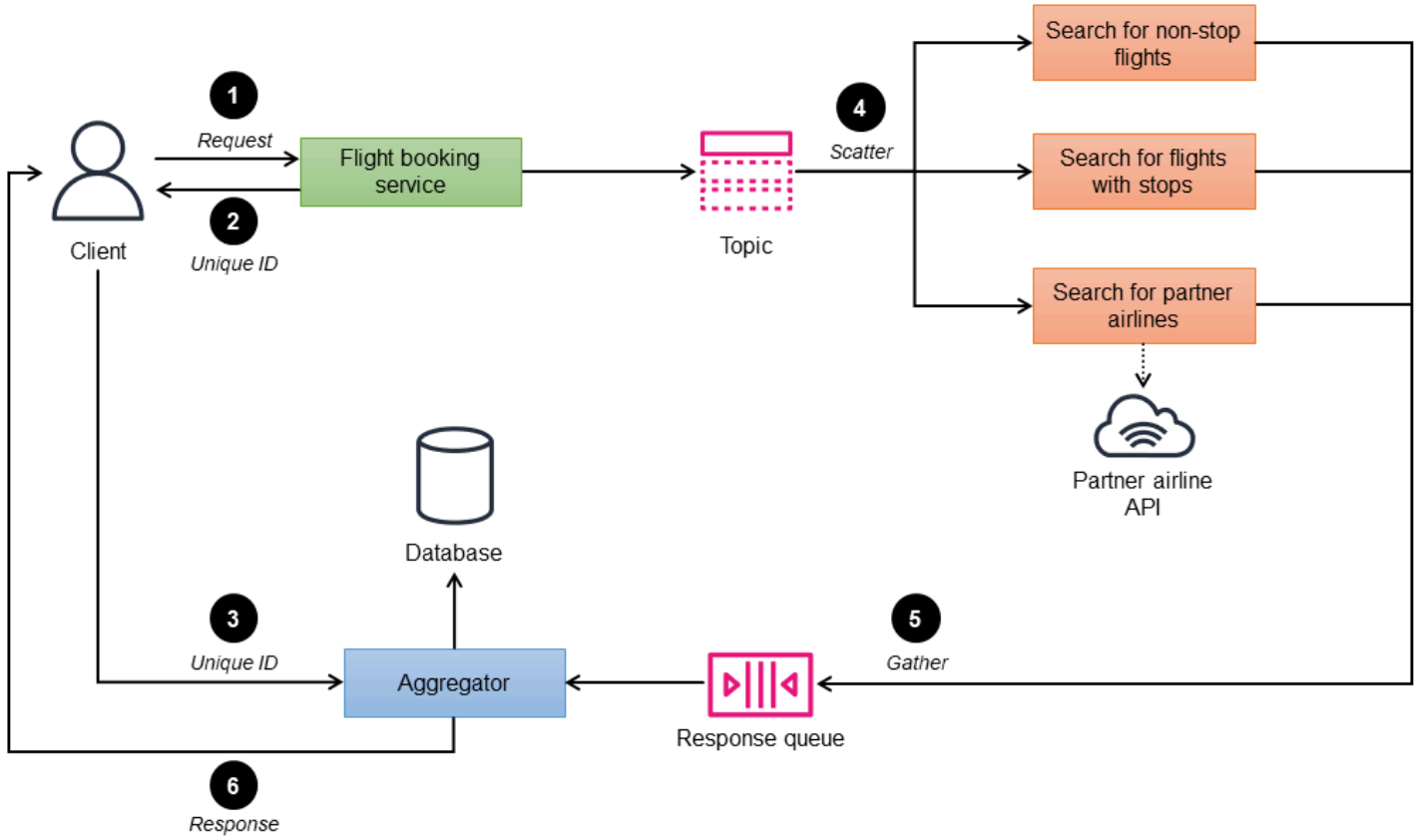
Pengontrol (prosesor file data) mengatur seluruh rangkaian pemanggilan, dan mengetahui semua titik akhir pemesanan untuk dipanggil. Ini dapat mengonfigurasi parameter batas waktu untuk mengabaikan respons yang memakan waktu terlalu lama. Ketika permintaan telah dikirim, agregator menunggu tanggapan kembali dari setiap titik akhir. Untuk menerapkan ketahanan, setiap layanan mikro dapat digunakan dengan beberapa instance untuk penyeimbangan beban. Agregator mendapatkan hasilnya, menggabungkannya menjadi satu pesan respons, dan menghapus data duplikat sebelum diproses lebih lanjut. Tanggapan bahwa time out diabaikan. Pengontrol juga dapat bertindak sebagai agregator alih-alih menggunakan layanan agregator terpisah.

Scatter dengan lelang

Jika pengontrol tidak mengetahui penerima atau penerima digabungkan secara longgar, Anda dapat menggunakan metode pencar dengan lelang. Dalam metode ini, penerima berlangganan topik dan pengontrol menerbitkan permintaan ke topik tersebut. Penerima mempublikasikan hasil ke antrian respons. Karena pengontrol root tidak mengetahui penerima, proses pengumpulan menggunakan agregator (pola pesan lain) untuk mengumpulkan tanggapan dan menyaringnya menjadi satu pesan respons. Agregator menggunakan ID unik untuk mengidentifikasi sekelompok permintaan.

Misalnya, dalam diagram berikut, metode pencar dengan lelang digunakan untuk menerapkan layanan pemesanan penerbangan untuk situs web maskapai penerbangan. Situs web ini memungkinkan pengguna untuk mencari dan menampilkan penerbangan dari maskapai penerbangan sendiri dan operator mitranya, dan harus menampilkan status pencarian secara real time. Layanan pemesanan penerbangan terdiri dari tiga layanan mikro pencarian: penerbangan

non-stop, penerbangan dengan pemberhentian, dan maskapai mitra. Pencarian maskapai mitra memanggil titik akhir API mitra untuk mendapatkan tanggapan.



1. Layanan pemesanan penerbangan (pengontrol) mengambil kriteria pencarian sebagai masukan dari klien, dan memproses serta menerbitkan permintaan ke topik.
2. Pengontrol menggunakan ID unik untuk mengidentifikasi setiap kelompok permintaan.
3. Klien mengirimkan ID unik ke agregator untuk langkah 6.
4. Layanan mikro pencarian pemesanan yang telah berlangganan topik pemesanan menerima permintaan.
5. Layanan mikro memproses permintaan dan mengembalikan ketersediaan kursi untuk kriteria pencarian yang diberikan ke antrian respons.
6. Agregator mengumpulkan semua pesan respons yang disimpan dalam database sementara, mengelompokkan penerbangan dengan ID unik, membuat respons terpadu tunggal, dan mengirimkannya kembali ke klien.

Implementasi menggunakan Layanan AWS

Sebarkan dengan distribusi

Dalam arsitektur berikut, pengontrol root adalah prosesor file data (Amazon ECS) yang membagi data permintaan masuk menjadi bucket Amazon Simple Storage Service (Amazon S3) individual dan memulai alur kerja. AWS Step Functions Alur kerja mengunduh data dan memulai pemrosesan file paralel. `Parallel`Negara menunggu semua tugas untuk mengembalikan respons. Suatu AWS Lambda fungsi mengumpulkan data dan menyimpannya kembali ke Amazon S3.

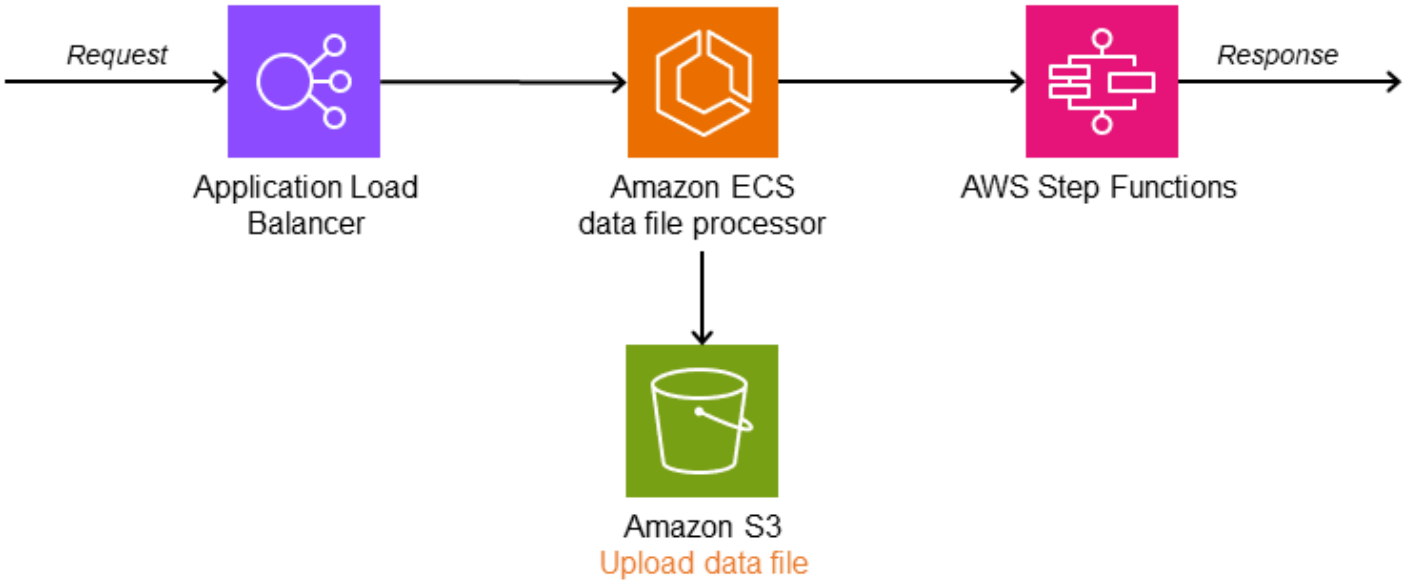
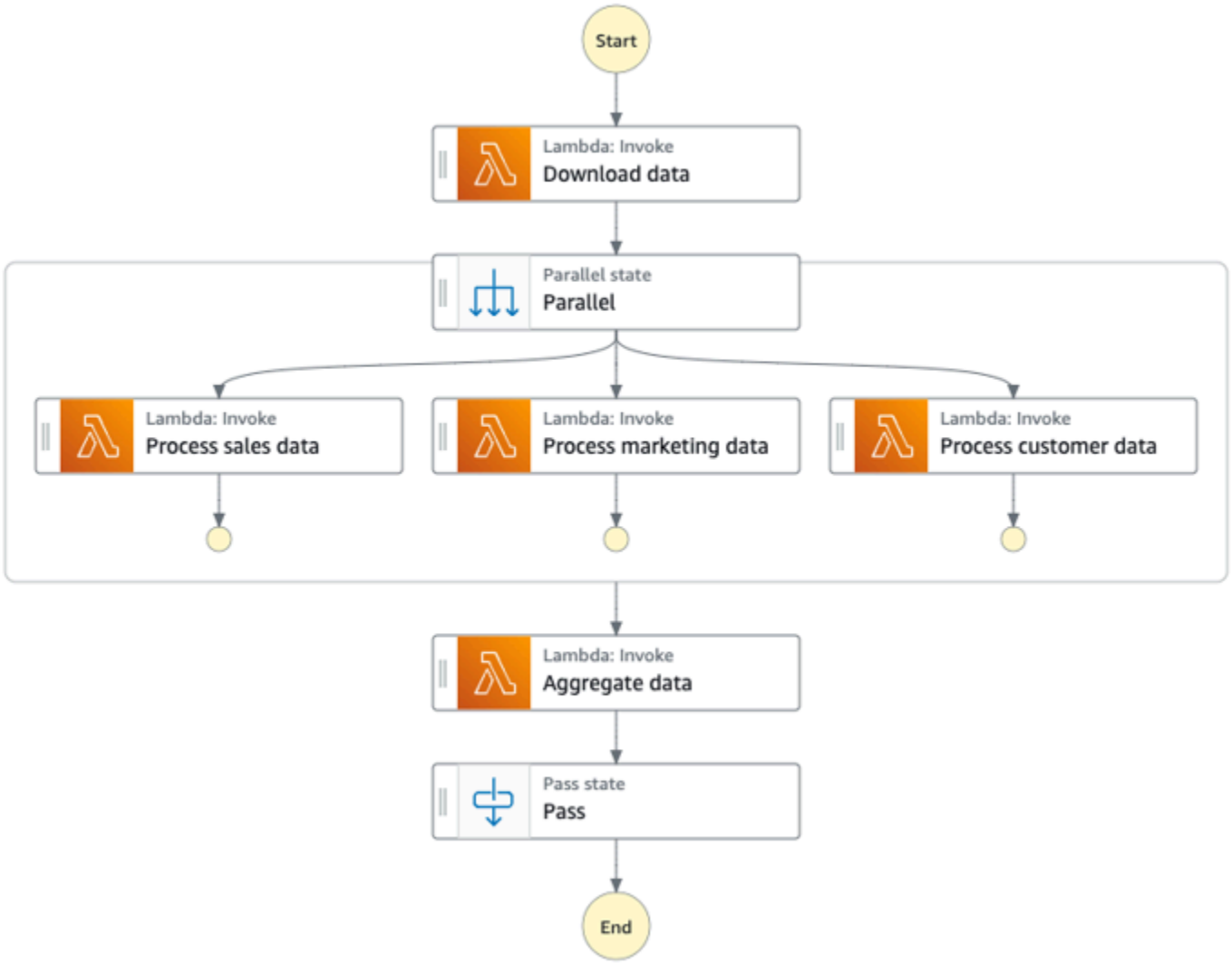
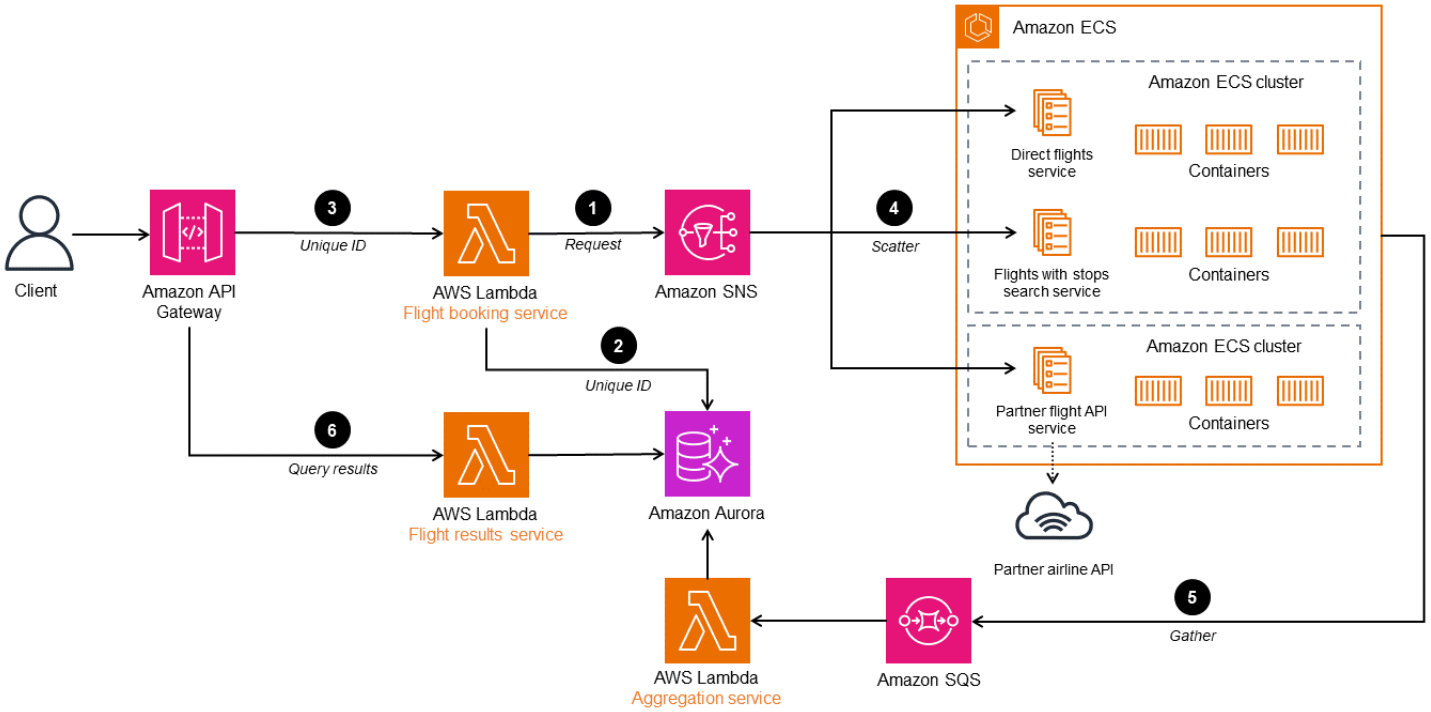


Diagram berikut menggambarkan alur kerja Step Functions dengan status. `Parallel`



Scatter dengan lelang

Diagram berikut menunjukkan AWS arsitektur untuk pencar dengan metode lelang. Layanan pemesanan penerbangan root controller menyebarkan permintaan pencarian penerbangan ke beberapa layanan mikro. Saluran berlangganan publikasi diimplementasikan dengan Amazon Simple Notification Service (Amazon SNS), yang merupakan layanan pesan terkelola untuk komunikasi. Amazon SNS mendukung pesan antara aplikasi microservice yang dipisahkan atau komunikasi langsung ke pengguna. Anda dapat menerapkan layanan mikro penerima di Amazon Elastic Kubernetes Service (Amazon EKS) atau Amazon Elastic Container Service (Amazon ECS) Service Elastic Container (Amazon ECS) untuk pengelolaan dan skalabilitas yang lebih baik. Layanan hasil penerbangan mengembalikan hasilnya kepada klien. Ini dapat diimplementasikan di AWS Lambda atau layanan orkestrasi kontainer lainnya seperti Amazon ECS atau Amazon EKS.



1. Layanan pemesanan penerbangan (controller) mengambil kriteria pencarian sebagai masukan dari klien, dan memproses serta menerbitkan permintaan ke topik SNS.
2. Pengontrol menerbitkan ID unik ke database Amazon Aurora untuk mengidentifikasi permintaan.
3. Klien mengirimkan ID unik ke klien untuk langkah 6.
4. Layanan mikro pencarian pemesanan yang telah berlangganan topik pemesanan menerima permintaan.
5. Layanan mikro memproses permintaan dan mengembalikan ketersediaan kursi untuk kriteria penelusuran yang diberikan ke antrian respons di Amazon Simple Queue Service (Amazon SQS). Agregator mengumpulkan semua pesan respons dan menyimpannya dalam database sementara.
6. Layanan hasil penerbangan mengelompokkan penerbangan dengan ID unik, menciptakan respons terpadu tunggal, dan mengirimkannya kembali ke klien.

Jika Anda ingin menambahkan pencarian maskapai penerbangan lain ke arsitektur ini, Anda menambahkan layanan mikro yang berlangganan topik SNS dan menerbitkan ke antrean SQS.

Untuk meringkas, pola scatter-gathering memungkinkan sistem terdistribusi untuk mencapai paralelisasi yang efisien, mengurangi latensi, dan menangani komunikasi asinkron dengan mulus.

GitHub repositori

Untuk implementasi lengkap arsitektur sampel untuk pola ini, lihat GitHub repositori di https://github.com/aws-samples/asynchronous-messaging-workshop/tree/master/code/lab-3.

Lokakarya

- Laboratorium Scatter-gathering di bengkel Layanan Mikro Decoupled

Referensi blog

- Pola integrasi aplikasi untuk layanan mikro

Konten terkait

- Pola terbitkan-berlangganan

Pola ara pencekik

Niat

Pola ara pencekik membantu memigrasikan aplikasi monolitik ke arsitektur layanan mikro secara bertahap, dengan pengurangan risiko transformasi dan gangguan bisnis.

Motivasi

Aplikasi monolitik dikembangkan untuk menyediakan sebagian besar fungsinya dalam satu proses atau wadah. Kode digabungkan dengan erat. Akibatnya, perubahan aplikasi memerlukan pengujian ulang menyeluruh untuk menghindari masalah regresi. Perubahan tidak dapat diuji secara terpisah, yang berdampak pada waktu siklus. Karena aplikasi diperkaya dengan lebih banyak fitur, kompleksitas tinggi dapat menyebabkan lebih banyak waktu yang dihabiskan untuk pemeliharaan, peningkatan waktu ke pasar, dan, akibatnya, inovasi produk lambat.

Ketika skala aplikasi dalam ukuran, itu meningkatkan beban kognitif pada tim dan dapat menyebabkan batas kepemilikan tim yang tidak jelas. Penskalaan fitur individual berdasarkan beban tidak mungkin—seluruh aplikasi harus diskalakan untuk mendukung beban puncak. Seiring bertambahnya usia sistem, teknologi dapat menjadi usang, yang menaikkan biaya dukungan. Aplikasi monolitik dan warisan mengikuti praktik terbaik yang tersedia pada saat pengembangan dan tidak dirancang untuk didistribusikan.

Ketika aplikasi monolitik dimigrasikan ke arsitektur layanan mikro, itu dapat dibagi menjadi komponen yang lebih kecil. Komponen-komponen ini dapat diskalakan secara independen, dapat dirilis secara independen, dan dapat dimiliki oleh tim individu. Ini menghasilkan kecepatan perubahan yang lebih tinggi, karena perubahan dilokalisasi dan dapat diuji dan dilepaskan dengan cepat. Perubahan memiliki cakupan dampak yang lebih kecil karena komponen digabungkan secara longgar dan dapat digunakan secara individual.

Mengganti monolit sepenuhnya dengan aplikasi layanan mikro dengan menulis ulang atau memfaktorkan ulang kode adalah usaha besar dan risiko besar. Migrasi big bang, di mana monolit bermigrasi dalam satu operasi, memperkenalkan risiko transformasi dan gangguan bisnis. Sementara aplikasi sedang difaktorkan ulang, sangat sulit atau bahkan tidak mungkin untuk menambahkan fitur baru.

Salah satu cara untuk mengatasi masalah ini adalah dengan menggunakan pola ara pencekik, yang diperkenalkan oleh Martin Fowler. Pola ini melibatkan pindah ke layanan mikro dengan secara

bertahap mengekstraksi fitur dan membuat aplikasi baru di sekitar sistem yang ada. Fitur dalam monolit digantikan oleh layanan mikro secara bertahap, dan pengguna aplikasi dapat menggunakan fitur yang baru dimigrasi secara progresif. Ketika semua fitur dipindahkan ke sistem baru, aplikasi monolitik dapat dinonaktifkan dengan aman.

Penerapan

Gunakan pola ara pencekik saat:

- Anda ingin memigrasikan aplikasi monolitik Anda secara bertahap ke arsitektur layanan mikro.
- Pendekatan migrasi big bang berisiko karena ukuran dan kompleksitas monolit.
- Bisnis ingin menambahkan fitur baru dan tidak sabar menunggu transformasi selesai.
- Pengguna akhir harus terkena dampak minimal selama transformasi.

Masalah dan pertimbangan

- Akses basis kode: Untuk mengimplementasikan pola ara pencekik, Anda harus memiliki akses ke basis kode aplikasi monolit. Karena fitur dimigrasikan keluar dari monolit, Anda perlu membuat perubahan kode kecil dan menerapkan lapisan anti-korupsi di dalam monolit untuk merutekan panggilan ke layanan mikro baru. Anda tidak dapat mencegat panggilan tanpa akses basis kode. Akses basis kode juga penting untuk mengarahkan permintaan yang masuk – beberapa refactoring kode mungkin diperlukan sehingga lapisan proxy dapat mencegat panggilan untuk fitur yang dimigrasi dan merutekkannya ke layanan mikro.
- Domain tidak jelas: Dekomposisi prematur sistem bisa mahal, terutama ketika domain tidak jelas, dan mungkin saja batas layanan salah. Domain-driven design (DDD) adalah mekanisme untuk memahami domain, dan event storming adalah teknik untuk menentukan batas domain.
- Mengidentifikasi layanan mikro: Anda dapat menggunakan DDD sebagai alat utama untuk mengidentifikasi layanan mikro. Untuk mengidentifikasi layanan mikro, cari pembagian alami antara kelas layanan. Banyak layanan akan memiliki objek akses data mereka sendiri dan akan memisahkan dengan mudah. Layanan yang memiliki logika bisnis terkait dan kelas yang tidak memiliki atau sedikit dependensi adalah kandidat yang baik untuk layanan mikro. Anda dapat memfaktorkan ulang kode sebelum memecah monolit untuk mencegah kopling yang ketat. Anda juga harus mempertimbangkan persyaratan kepatuhan, irama rilis, lokasi geografis tim, kebutuhan penskalaan, penggunaan kebutuhan teknologi berbasis kasus, dan beban kognitif tim.

- Lapisan anti-korupsi: Selama proses migrasi, ketika fitur dalam monolit harus memanggil fitur yang dimigrasikan sebagai layanan mikro, Anda harus menerapkan lapisan anti-korupsi (ACL) yang merutekan setiap panggilan ke layanan mikro yang sesuai. Untuk memisahkan dan mencegah perubahan pada penelepon yang ada di dalam monolit, ACL berfungsi sebagai adaptor atau fasad yang mengubah panggilan ke antarmuka yang lebih baru. Ini dibahas secara rinci di bagian Implementasi pola ACL sebelumnya dalam panduan ini.
- Kegagalan lapisan proxy: Selama migrasi, lapisan proxy mencegat permintaan yang masuk ke aplikasi monolitik dan merutekkannya ke sistem lama atau sistem baru. Namun, lapisan proxy ini dapat menjadi satu titik kegagalan atau kemacetan kinerja.
- Kompleksitas aplikasi: Monolit besar paling diuntungkan dari pola ara pencekik. Untuk aplikasi kecil, di mana kompleksitas refactoring lengkap rendah, mungkin lebih efisien untuk menulis ulang aplikasi dalam arsitektur layanan mikro daripada memigrasikannya.
- Interaksi layanan: Layanan mikro dapat berkomunikasi secara sinkron atau asinkron. Ketika komunikasi sinkron diperlukan, pertimbangkan apakah batas waktu dapat menyebabkan koneksi atau konsumsi kumpulan thread, yang mengakibatkan masalah kinerja aplikasi. Dalam kasus seperti itu, gunakan pola pemutus sirkuit untuk mengembalikan kegagalan langsung untuk operasi yang cenderung gagal untuk waktu yang lama. Komunikasi asinkron dapat dicapai dengan menggunakan acara dan antrian pesan.
- Agregasi data: Dalam arsitektur microservices, data didistribusikan di seluruh database. Ketika agregasi data diperlukan, Anda dapat menggunakan AWS AppSyncdi ujung depan, atau pola pemisahan tanggung jawab permintaan perintah (CQRS) di backend.
- Konsistensi data: Layanan mikro memiliki penyimpanan data mereka, dan aplikasi monolitik juga berpotensi menggunakan data ini. Untuk mengaktifkan berbagi, Anda dapat menyinkronkan penyimpanan data layanan mikro baru dengan database aplikasi monolitik dengan menggunakan antrian dan agen. Namun, ini dapat menyebabkan redundansi data dan konsistensi akhirnya antara dua penyimpanan data, jadi kami sarankan Anda memperlakukannya sebagai solusi taktis sampai Anda dapat membuat solusi jangka panjang seperti data lake.

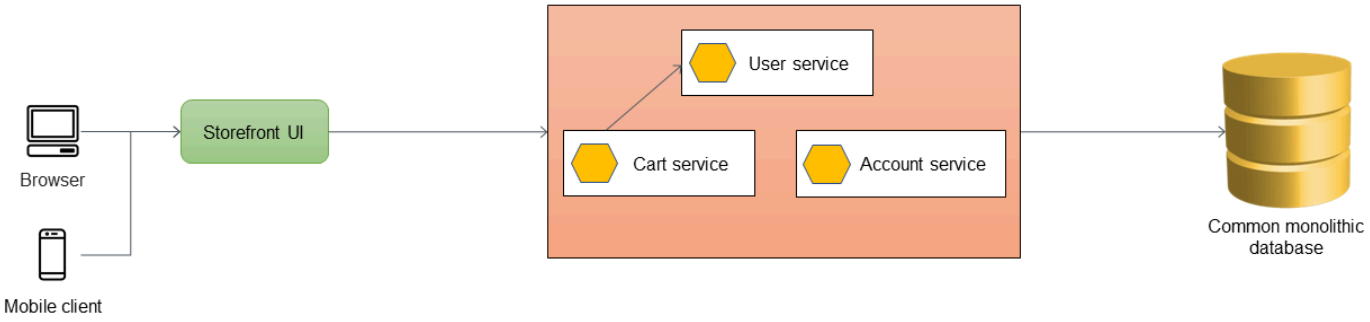
Implementasi

Dalam pola ara pencekik, Anda mengganti fungsionalitas tertentu dengan layanan atau aplikasi baru, satu komponen pada satu waktu. Lapisan proxy mencegat permintaan yang masuk ke aplikasi monolitik dan merutekan mereka ke sistem lama atau sistem baru. Karena lapisan proxy mengarahkan pengguna ke aplikasi yang benar, Anda dapat menambahkan fitur ke sistem baru

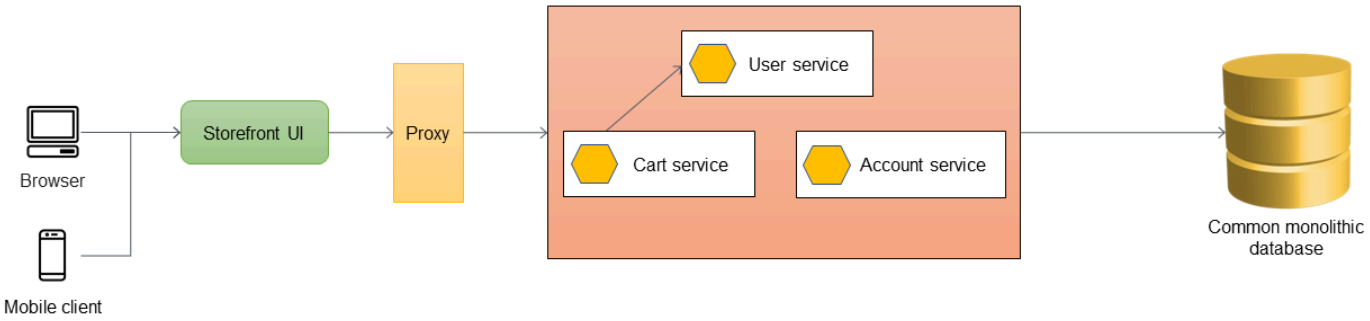
sambil memastikan bahwa monolit terus berfungsi. Sistem baru akhirnya menggantikan semua fitur dari sistem lama, dan Anda dapat menonaktifkannya.

Arsitektur tingkat tinggi

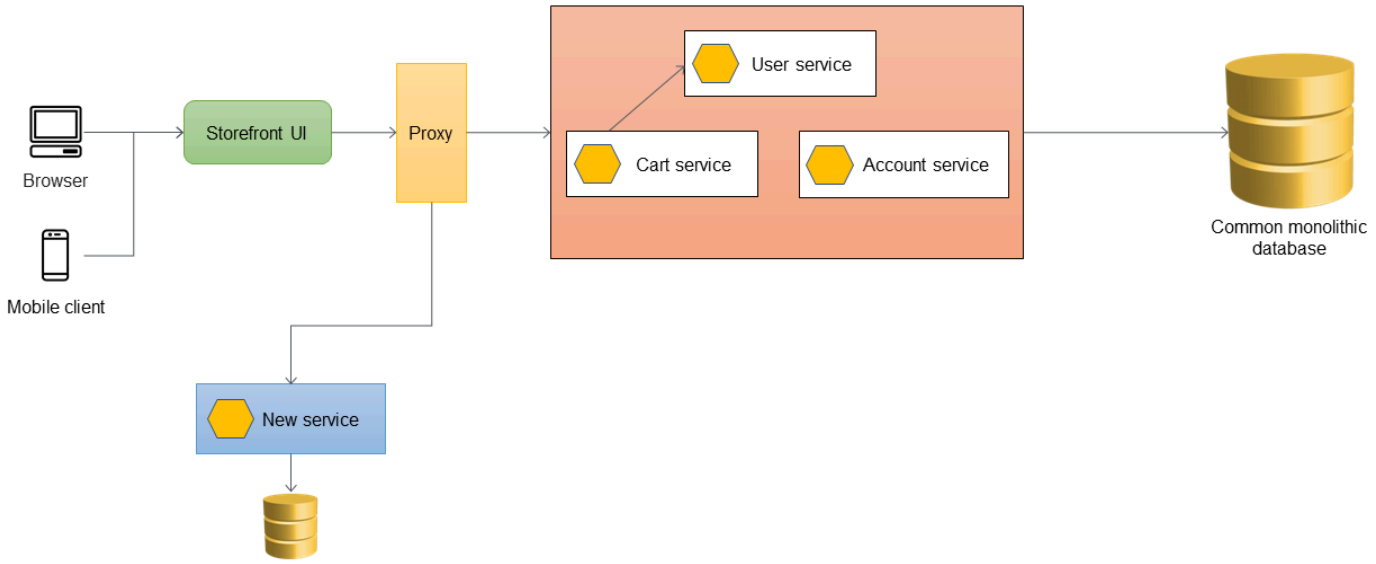
Dalam diagram berikut, aplikasi monolitik memiliki tiga layanan: layanan pengguna, layanan keranjang, dan layanan akun. Layanan keranjang tergantung pada layanan pengguna, dan aplikasi menggunakan database relasional monolitik.



Langkah pertama adalah menambahkan lapisan proxy antara UI etalase dan aplikasi monolitik. Pada awalnya, proxy merutekan semua lalu lintas ke aplikasi monolitik.

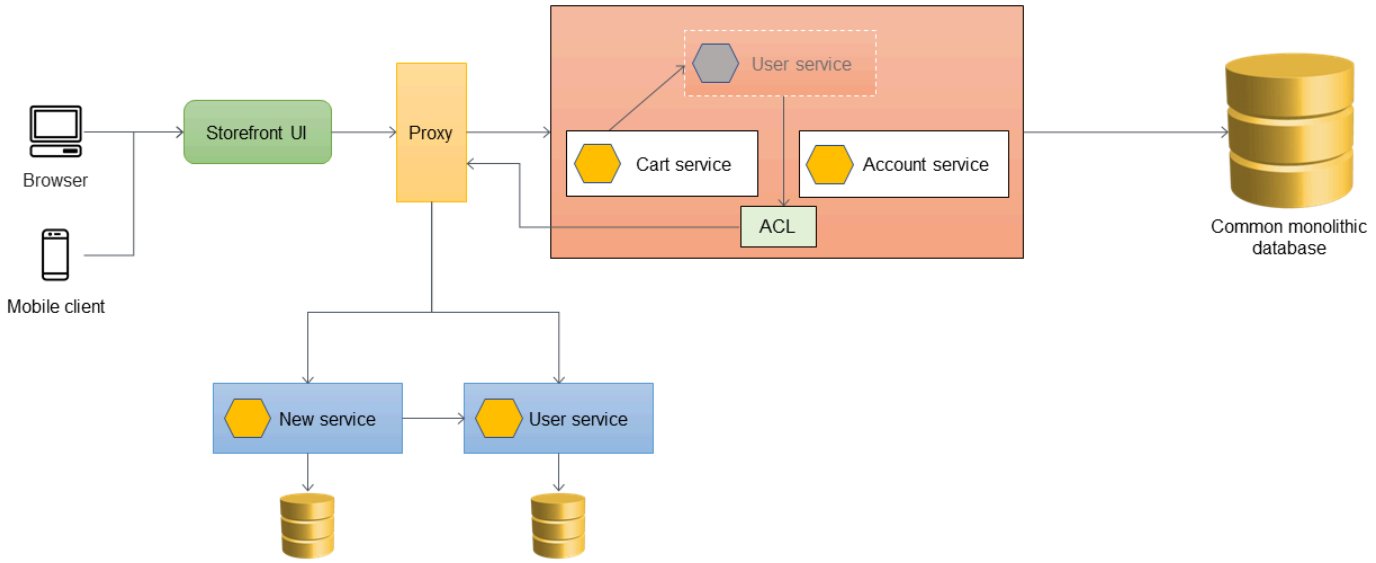


Ketika Anda ingin menambahkan fitur baru ke aplikasi Anda, Anda menerapkannya sebagai layanan mikro baru alih-alih menambahkan fitur ke monolit yang ada. Namun, Anda terus memperbaiki bug di monolit untuk memastikan stabilitas aplikasi. Dalam diagram berikut, lapisan proxy merutekan panggilan ke monolit atau ke layanan mikro baru berdasarkan URL API.



Menambahkan lapisan anti-korupsi

Dalam arsitektur berikut, layanan pengguna telah dimigrasikan ke layanan mikro. Layanan keranjang memanggil layanan pengguna, tetapi implementasinya tidak lagi tersedia dalam monolit. Selain itu, antarmuka layanan yang baru dimigrasi mungkin tidak cocok dengan antarmuka sebelumnya di dalam aplikasi monolitik. Untuk mengatasi perubahan ini, Anda menerapkan ACL. Selama proses migrasi, ketika fitur dalam monolit perlu memanggil fitur yang dimigrasi sebagai layanan mikro, ACL mengubah panggilan ke antarmuka baru dan merutekkannya ke layanan mikro yang sesuai.

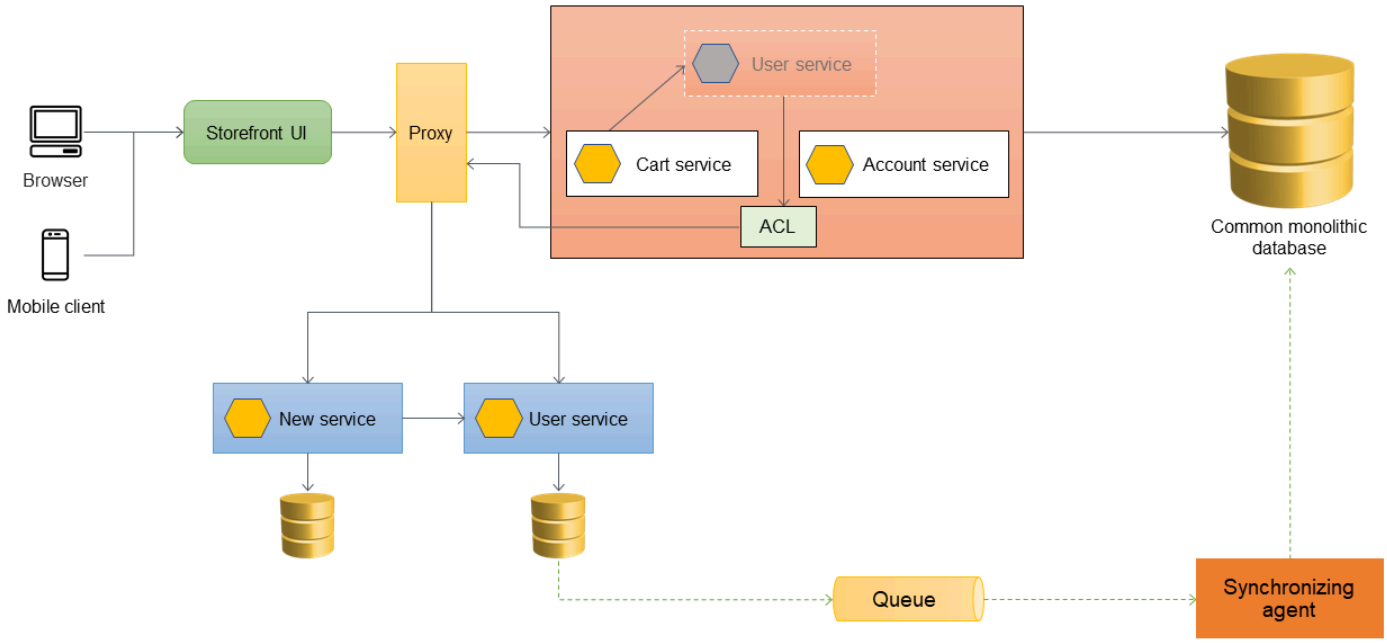


Anda dapat mengimplementasikan ACL di dalam aplikasi monolitik sebagai kelas yang khusus untuk layanan yang dimigrasi; misalnya, atau. `UserServiceFacade` `UserServiceAdapter` ACL harus dinonaktifkan setelah semua layanan dependen telah dimigrasikan ke arsitektur layanan mikro.

Saat Anda menggunakan ACL, layanan keranjang masih memanggil layanan pengguna dalam monolit, dan layanan pengguna mengalihkan panggilan ke layanan mikro melalui ACL. Layanan keranjang harus tetap memanggil layanan pengguna tanpa mengetahui migrasi layanan mikro. Kopling longgar ini diperlukan untuk mengurangi regresi dan gangguan bisnis.

Menangani sinkronisasi data

Sebagai praktik terbaik, layanan mikro harus memiliki datanya. Layanan pengguna menyimpan datanya di penyimpanan datanya sendiri. Mungkin perlu menyinkronkan data dengan database monolitik untuk menangani dependensi seperti pelaporan dan untuk mendukung aplikasi hilir yang belum siap untuk mengakses layanan mikro secara langsung. Aplikasi monolitik mungkin juga memerlukan data untuk fungsi dan komponen lain yang belum dimigrasikan ke layanan mikro. Jadi sinkronisasi data diperlukan antara microservice baru dan monolit. Untuk menyinkronkan data, Anda dapat memperkenalkan agen sinkronisasi antara layanan mikro pengguna dan database monolitik, seperti yang ditunjukkan pada diagram berikut. Layanan mikro pengguna mengirimkan acara ke antrian setiap kali database-nya diperbarui. Agen sinkronisasi mendengarkan antrian dan terus memperbarui database monolitik. Data dalam database monolitik pada akhirnya konsisten untuk data yang sedang disinkronkan.



Migrasi layanan tambahan

Ketika layanan keranjang dimigrasi keluar dari aplikasi monolitik, kodenya direvisi untuk memanggil layanan baru secara langsung, sehingga ACL tidak lagi merutekan panggilan tersebut. Diagram berikut menggambarkan arsitektur ini.

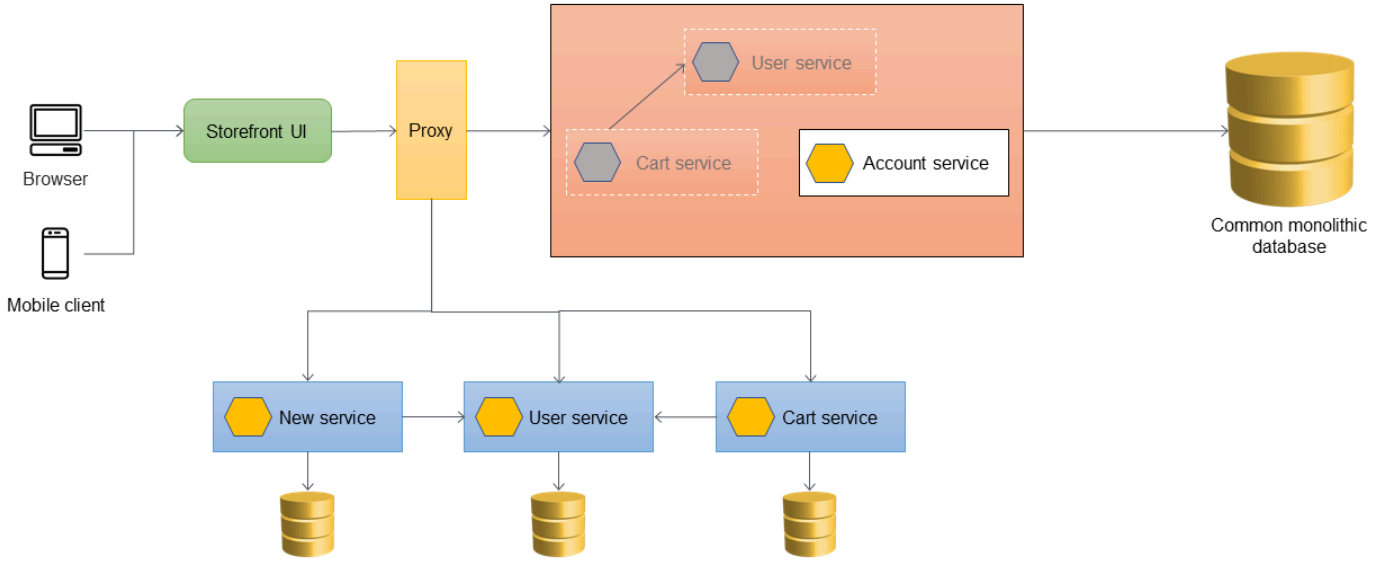


Diagram berikut menunjukkan keadaan tercekik terakhir di mana semua layanan telah dimigrasi keluar dari monolit dan hanya kerangka monolit yang tersisa. Data historis dapat dimigrasikan ke penyimpanan data yang dimiliki oleh layanan individual. ACL dapat dihilangkan, dan monolit siap dinonaktifkan pada tahap ini.

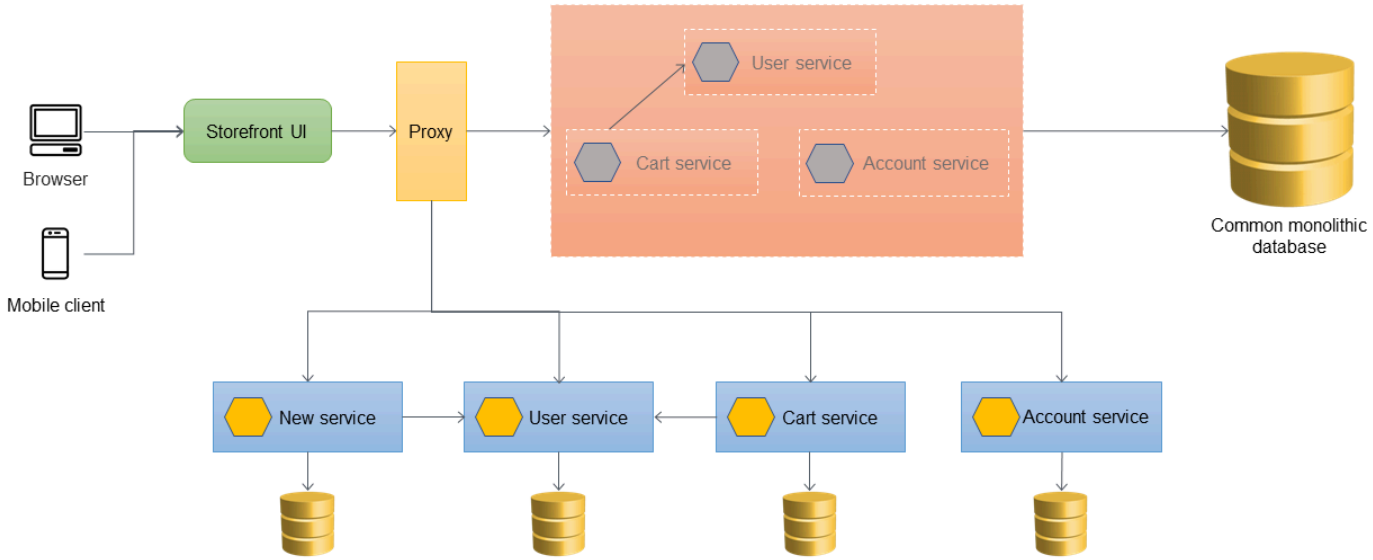
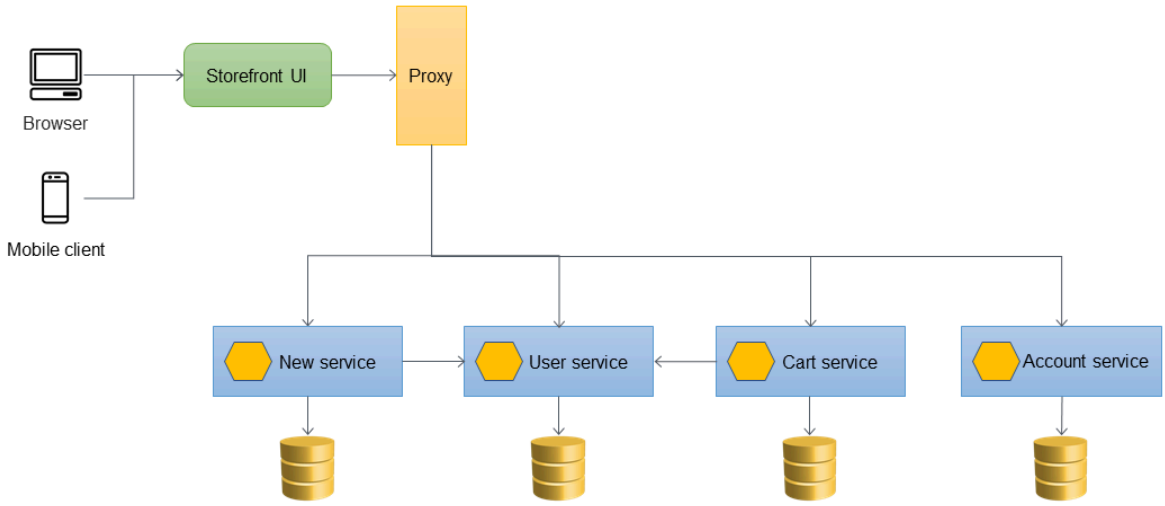


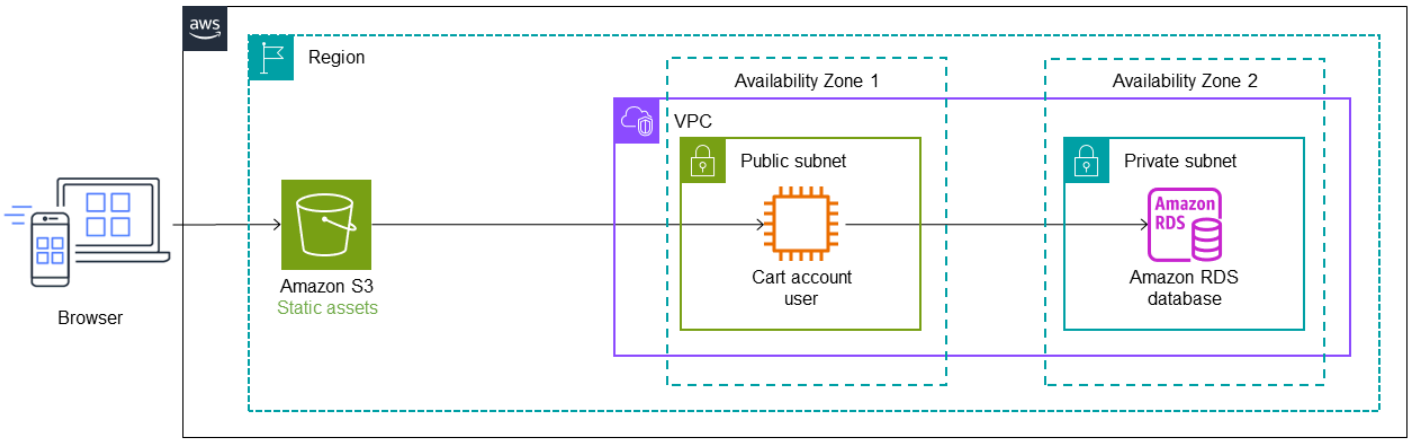
Diagram berikut menunjukkan arsitektur akhir setelah aplikasi monolitik dinonaktifkan. Anda dapat meng-host layanan mikro individu melalui URL berbasis sumber daya (seperti`http://www.storefront.com/user`) atau melalui domain mereka sendiri (misalnya,`http://user.storefront.com`) berdasarkan persyaratan aplikasi Anda. Untuk informasi selengkapnya tentang metode utama untuk mengekspos HTTP APIs ke konsumen hulu dengan menggunakan nama host dan jalur, lihat bagian pola perutean API.



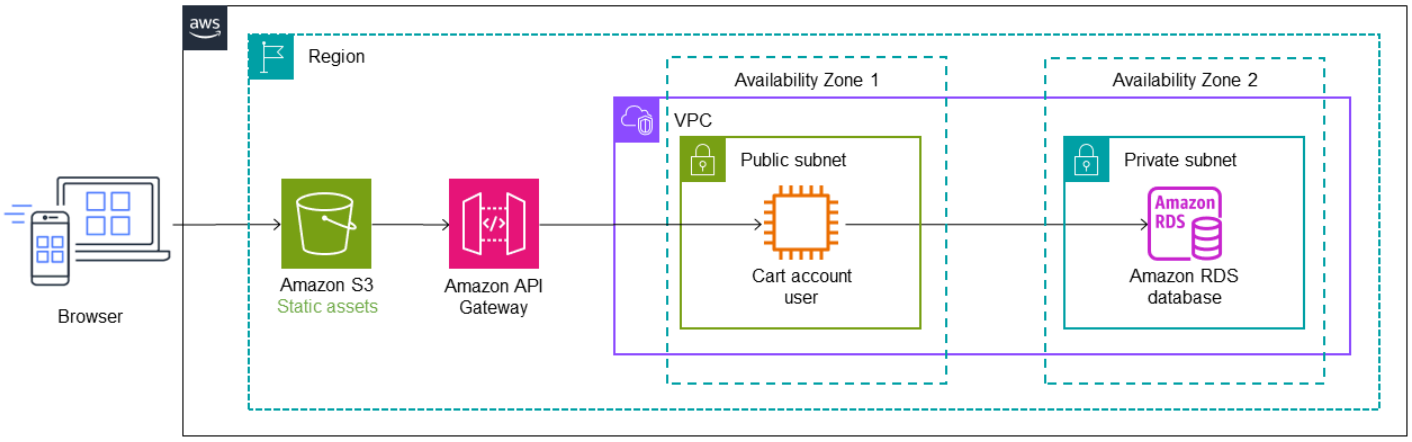
Implementasi menggunakan AWS layanan

Menggunakan API Gateway sebagai proxy aplikasi

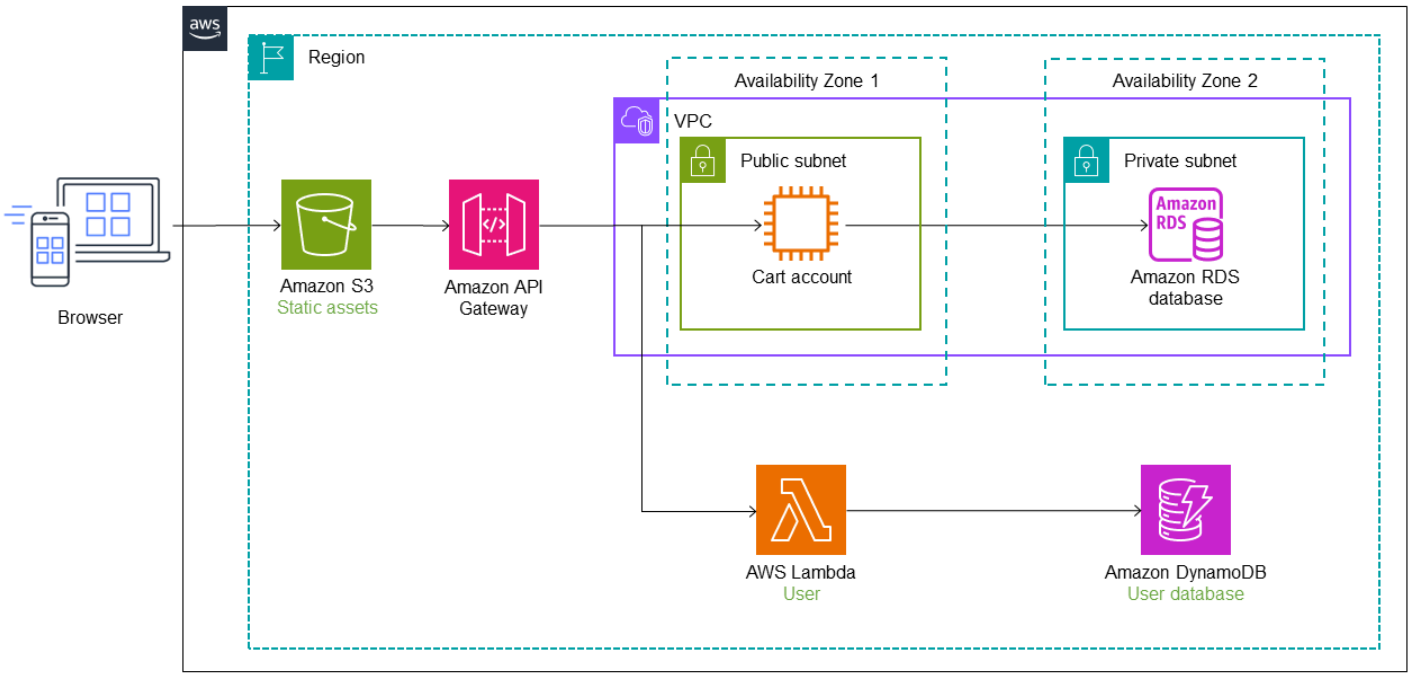
Diagram berikut menunjukkan keadaan awal aplikasi monolitik. Mari kita asumsikan bahwa itu dimigrasikan ke AWS dengan menggunakan lift-and-shift strategi, sehingga berjalan pada instance Amazon Elastic Compute Cloud (Amazon EC2) dan menggunakan database Amazon Relational Database Service (Amazon RDS). Untuk kesederhanaan, arsitektur menggunakan cloud pribadi virtual tunggal (VPC) dengan satu subnet pribadi dan satu publik, dan mari kita asumsikan bahwa layanan mikro awalnya akan digunakan dalam hal yang sama. Akun AWS(Praktik terbaik dalam lingkungan produksi adalah menggunakan arsitektur multi-akun untuk memastikan independensi penerapan.) EC2 Instance berada di Availability Zone tunggal di subnet publik, dan instance RDS berada di Availability Zone tunggal di subnet pribadi. Amazon Simple Storage Service (Amazon S3) menyimpan aset statis seperti JavaScript file, CSS, dan React untuk situs web.



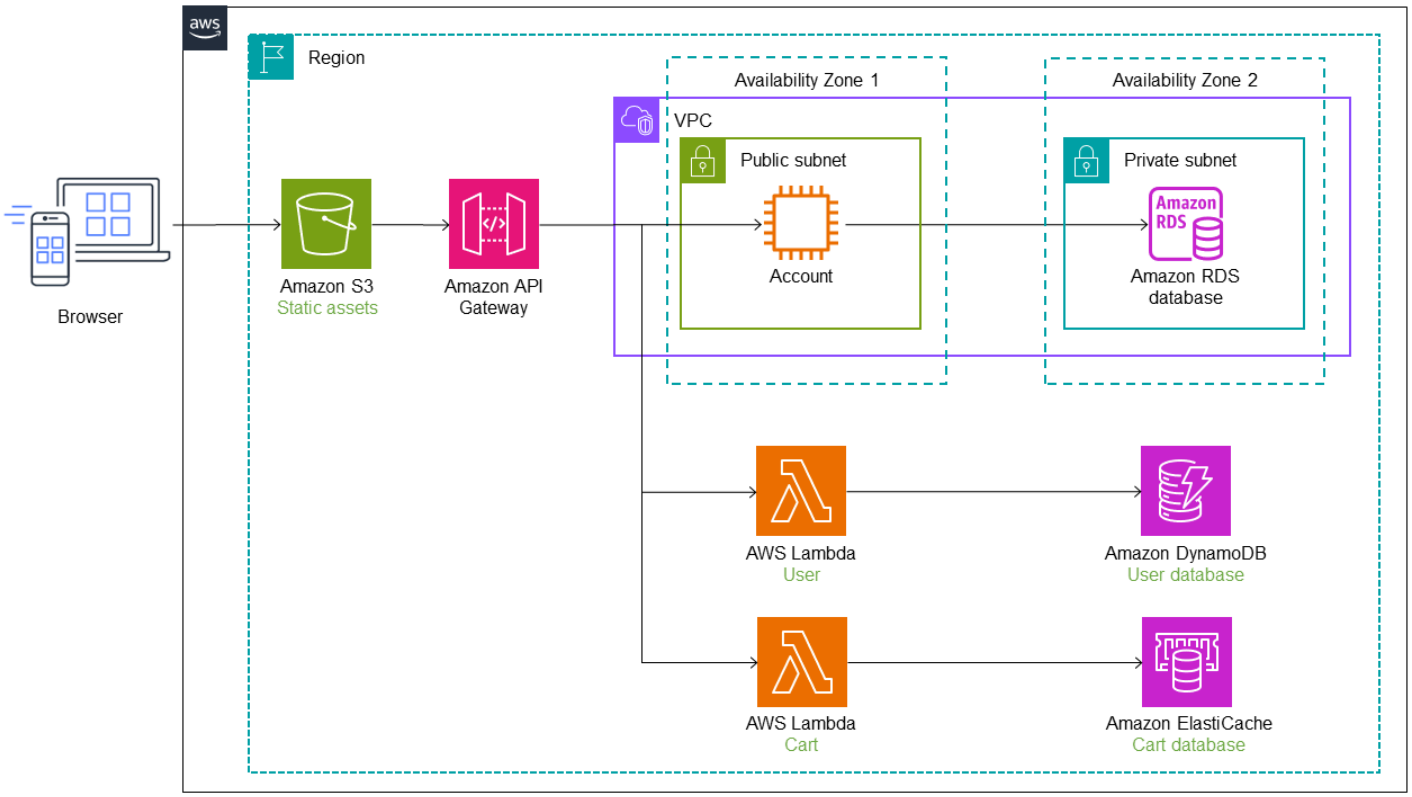
Dalam arsitektur berikut, AWS Migration Hub Refactor Spacesgunakan Amazon API Gateway di depan aplikasi monolitik. Refactor Spaces membuat infrastruktur refactoring di dalam akun Anda, dan API Gateway bertindak sebagai lapisan proxy untuk merutekan panggilan ke monolit. Awalnya, semua panggilan dialihkan ke aplikasi monolitik melalui lapisan proxy. Seperti dibahas sebelumnya, lapisan proxy dapat menjadi satu titik kegagalan. Namun, menggunakan API Gateway sebagai proxy mengurangi risiko karena ini adalah layanan multi-AZ tanpa server.



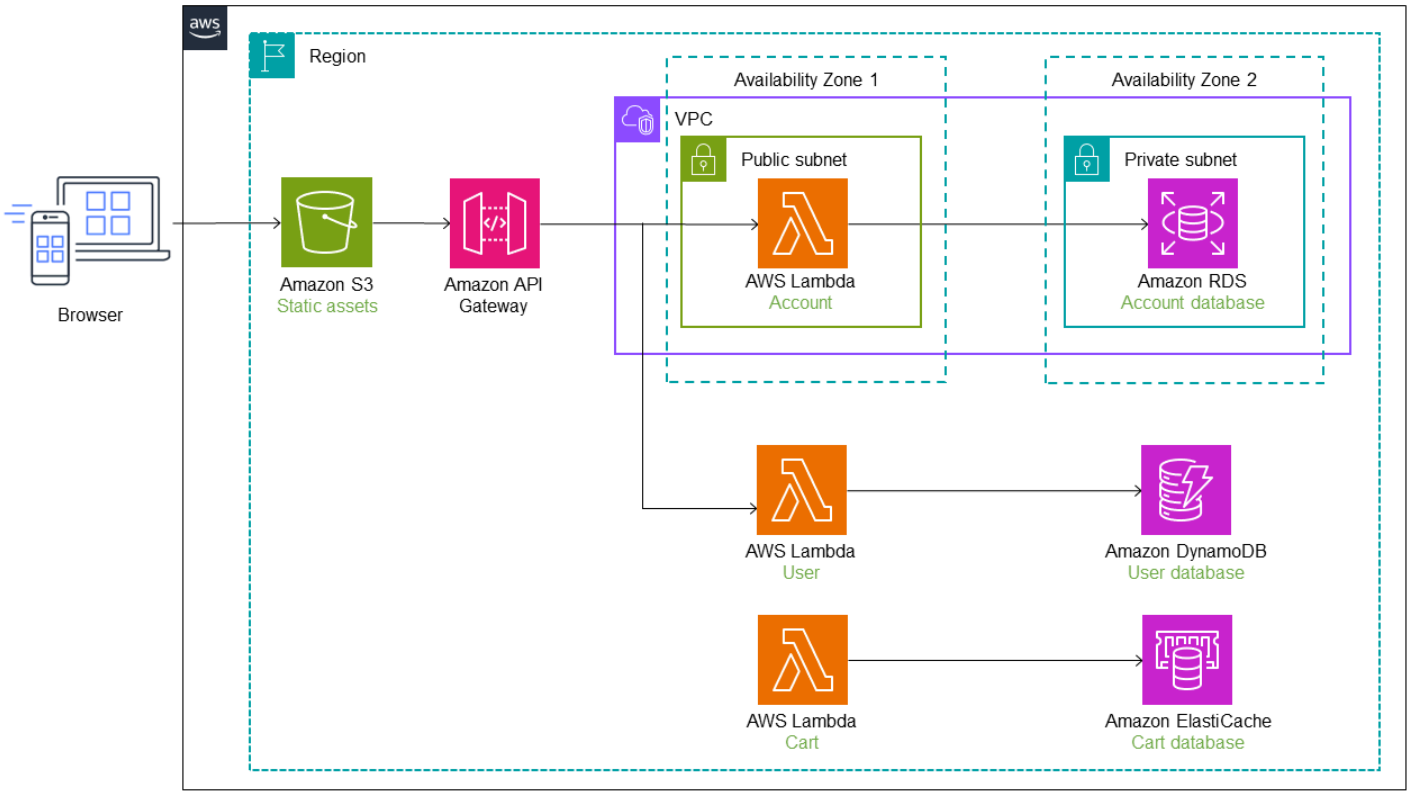
Layanan pengguna dimigrasikan ke fungsi Lambda, dan database Amazon DynamoDB menyimpan datanya. Titik akhir layanan Lambda dan rute default ditambahkan ke Refactor Spaces, dan API Gateway secara otomatis dikonfigurasi untuk merutekan panggilan ke fungsi Lambda.



Dalam diagram berikut, layanan keranjang juga telah dimigrasi keluar dari monolit dan ke fungsi Lambda. Rute tambahan dan titik akhir layanan ditambahkan ke Ruang Refactor, dan lalu lintas secara otomatis memotong ke fungsi Lambda`Cart`. Penyimpanan data untuk fungsi Lambda dikelola oleh Amazon. ElastiCache Aplikasi monolitik masih tetap dalam EC2 instance bersama dengan database Amazon RDS.



Pada diagram berikutnya, layanan terakhir (akun) dimigrasikan keluar dari monolit ke fungsi Lambda. Itu terus menggunakan database Amazon RDS asli. Arsitektur baru sekarang memiliki tiga microservices dengan database terpisah. Setiap layanan menggunakan jenis database yang berbeda. Konsep menggunakan database yang dibangun khusus untuk memenuhi kebutuhan spesifik layanan mikro disebut persistensi polyglot. Fungsi Lambda juga dapat diimplementasikan dalam bahasa pemrograman yang berbeda, sebagaimana ditentukan oleh kasus penggunaan. Selama refactoring, Refactor Spaces mengotomatiskan cutover dan routing lalu lintas ke Lambda. Ini menghemat waktu pembangun Anda untuk merancang, menyebarkan, dan mengkonfigurasi infrastruktur perutean.



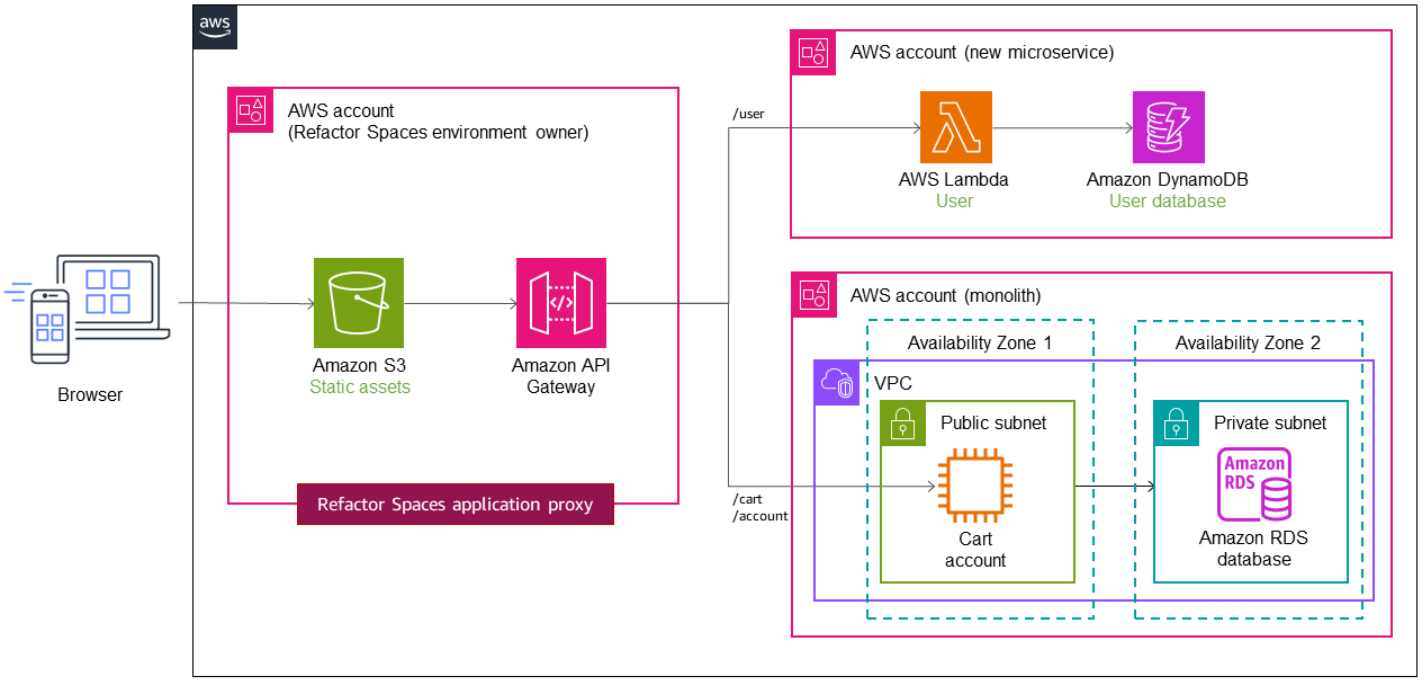
Menggunakan beberapa akun

Dalam implementasi sebelumnya, kami menggunakan VPC tunggal dengan subnet pribadi dan publik untuk aplikasi monolitik, dan kami menerapkan layanan mikro dalam hal yang sama demi kesederhanaan. Akun AWS Namun, ini jarang terjadi dalam skenario dunia nyata, di mana layanan mikro sering digunakan dalam beberapa Akun AWS untuk independensi penyebaran. Dalam struktur multi-akun, Anda perlu mengonfigurasi lalu lintas perutean dari monolit ke layanan baru di akun yang berbeda.

Refactor Spaces membantu Anda membuat dan mengonfigurasi AWS infrastruktur untuk merutekan panggilan API dari aplikasi monolitik. Refactor Spaces mengatur kebijakan API Gateway, Network Load Balancer, dan AWS Identity and Access Management berbasis sumber daya (IAM) di dalam akun Anda sebagai bagian dari sumber daya aplikasinya. AWS Anda dapat menambahkan layanan baru secara transparan dalam satu Akun AWS atau beberapa akun ke titik akhir HTTP eksternal. Semua sumber daya ini diatur di dalam Anda Akun AWS dan dapat disesuaikan dan dikonfigurasi setelah penerapan.

Mari kita asumsikan bahwa layanan pengguna dan keranjang dikerahkan ke dua akun yang berbeda, seperti yang ditunjukkan pada diagram berikut. Saat Anda menggunakan Refactor Spaces, Anda hanya perlu mengonfigurasi titik akhir layanan dan rute. Refactor Spaces mengotomatiskan integrasi

API Gateway—Lambda dan pembuatan kebijakan sumber daya Lambda, sehingga Anda dapat fokus pada refactoring layanan yang aman di luar monolit.



Untuk tutorial video tentang menggunakan Refactor Spaces, lihat Refactor Apps Incrementally with. AWS Migration Hub Refactor Spaces

Lokakarya

- Lokakarya modernisasi aplikasi berulang

Referensi blog

- AWS Migration Hub Refactor Spaces
- Deep Dive di sebuah AWS Migration Hub Refactor Spaces
- Arsitektur Referensi Pipelines Penerapan dan Implementasi Referensi

Konten terkait

- Pola perutean API
- Dokumentasi Ruang Refactor

Pola kotak keluar transaksional

Niat

Pola outbox transaksional menyelesaikan masalah operasi penulisan ganda yang terjadi dalam sistem terdistribusi ketika operasi tunggal melibatkan operasi penulisan database dan pesan atau pemberitahuan peristiwa. Operasi penulisan ganda terjadi ketika aplikasi menulis ke dua sistem yang berbeda; misalnya, ketika layanan mikro perlu menyimpan data dalam database dan mengirim pesan untuk memberi tahu sistem lain. Kegagalan dalam salah satu operasi ini dapat mengakibatkan data yang tidak konsisten.

Motivasi

Ketika microservice mengirimkan pemberitahuan peristiwa setelah pembaruan database, kedua operasi ini harus berjalan secara atomik untuk memastikan konsistensi dan keandalan data.

- Jika pembaruan database berhasil tetapi pemberitahuan acara gagal, layanan hilir tidak akan menyadari perubahan tersebut, dan sistem dapat memasukkan status yang tidak konsisten.
- Jika pembaruan database gagal tetapi pemberitahuan peristiwa dikirim, data bisa rusak, yang mungkin mempengaruhi keandalan sistem.

Penerapan

Gunakan pola outbox transaksional saat:

- Anda sedang membangun aplikasi berbasis peristiwa di mana pembaruan database memulai pemberitahuan peristiwa.
- Anda ingin memastikan atomisitas dalam operasi yang melibatkan dua layanan.
- Anda ingin menerapkan pola sumber acara.

Masalah dan pertimbangan

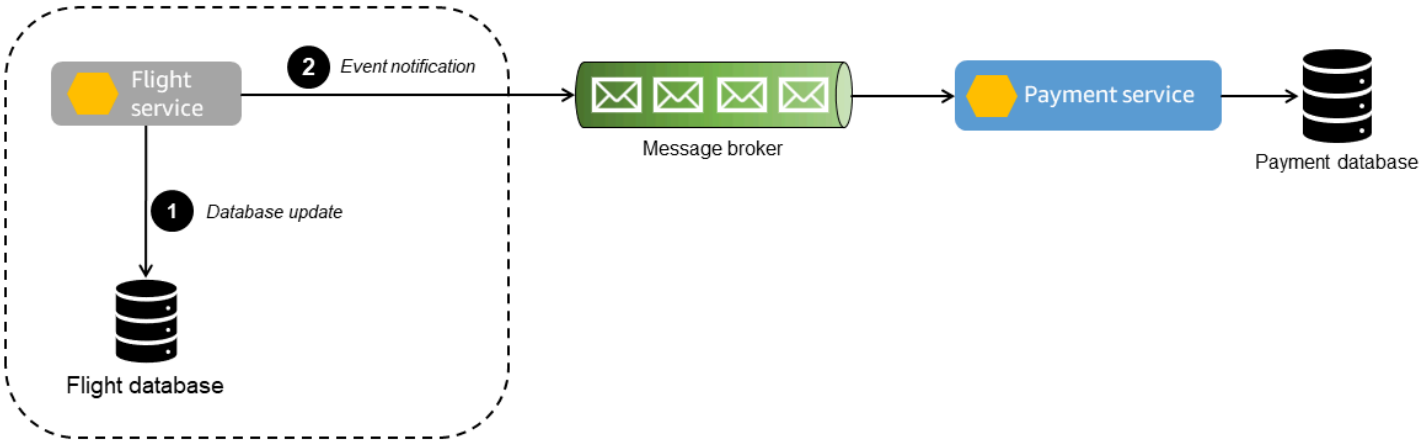
- Pesan duplikat: Layanan pemrosesan peristiwa mungkin mengirimkan pesan atau peristiwa duplikat, jadi sebaiknya Anda membuat layanan konsumsi idempoten dengan melacak pesan yang diproses.

- Urutan pemberitahuan: Kirim pesan atau acara dalam urutan yang sama di mana layanan memperbarui database. Ini sangat penting untuk pola sumber acara di mana Anda dapat menggunakan toko acara untuk point-in-time pemulihan penyimpanan data. Jika urutannya salah, mungkin mengganggu kualitas data. Konsistensi dan rollback database akhirnya dapat memperparah masalah jika urutan notifikasi tidak dipertahankan.
- Transaksi rollback: Jangan mengirimkan pemberitahuan peristiwa jika transaksi dibatalkan.
- Penanganan transaksi tingkat layanan: Jika transaksi mencakup layanan yang memerlukan pembaruan penyimpanan data, gunakan pola orkestrasi saga untuk menjaga integritas data di seluruh penyimpanan data.

Implementasi

Arsitektur tingkat tinggi

Diagram urutan berikut menunjukkan urutan peristiwa yang terjadi selama operasi penulisan ganda.



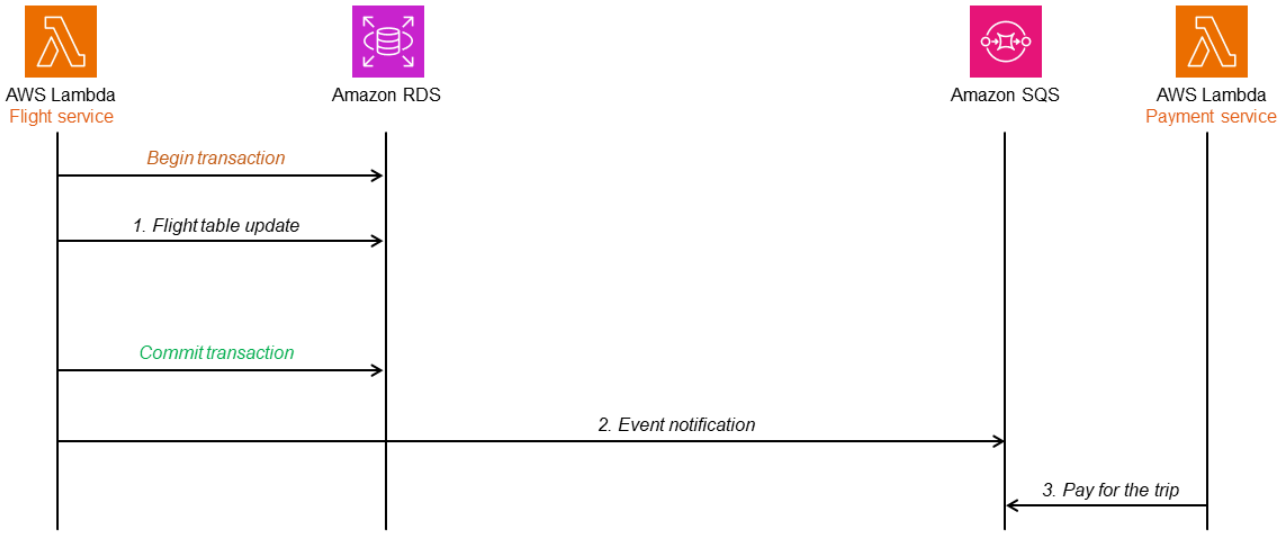
1. Layanan penerbangan menulis ke database dan mengirimkan pemberitahuan acara ke layanan pembayaran.
2. Broker pesan membawa pesan dan acara ke layanan pembayaran. Kegagalan apa pun di broker pesan mencegah layanan pembayaran menerima pembaruan.

Jika pembaruan database penerbangan gagal tetapi pemberitahuan dikirim, layanan pembayaran akan memproses pembayaran berdasarkan pemberitahuan acara. Ini akan menyebabkan inkonsistensi data hilir.

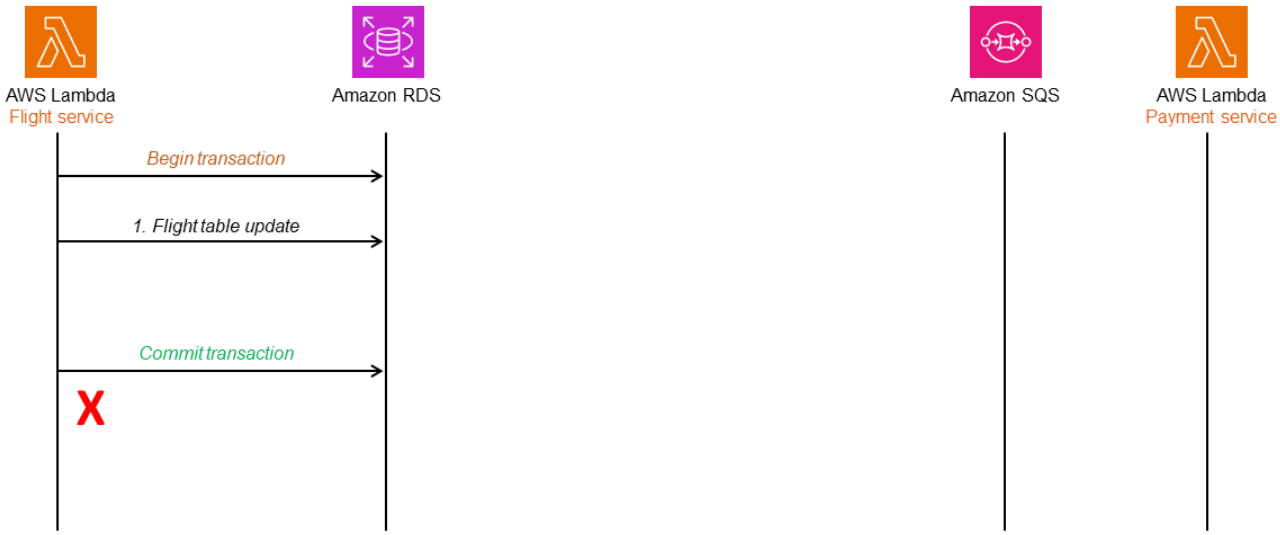
Implementasi menggunakan AWS layanan

Untuk mendemonstrasikan pola dalam diagram urutan, kami akan menggunakan AWS layanan berikut, seperti yang ditunjukkan pada diagram berikut.

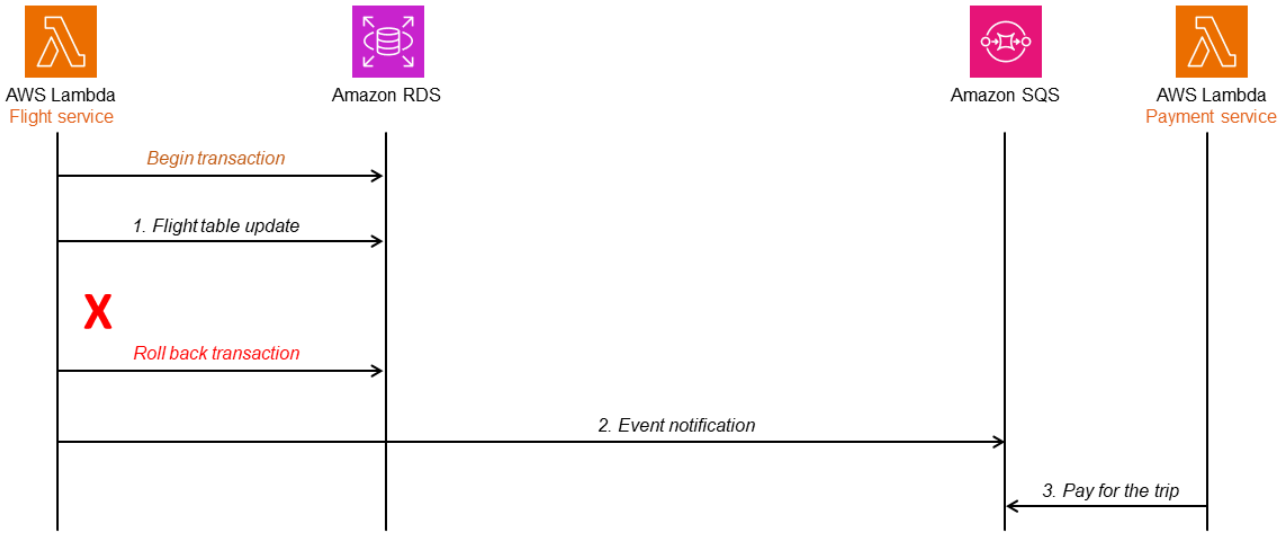
- Layanan mikro diimplementasikan dengan menggunakan AWS Lambda.
- Basis data utama dikelola oleh Amazon Relational Database Service (Amazon RDS).
- Amazon Simple Queue Service (Amazon Simple Queue Service) bertindak sebagai broker pesan yang menerima pemberitahuan acara.



Jika layanan penerbangan gagal setelah melakukan transaksi, hal ini dapat mengakibatkan pemberitahuan peristiwa tidak dikirim.



Namun, transaksi bisa gagal dan mundur, tetapi pemberitahuan acara mungkin masih dikirim, menyebabkan layanan pembayaran memproses pembayaran.



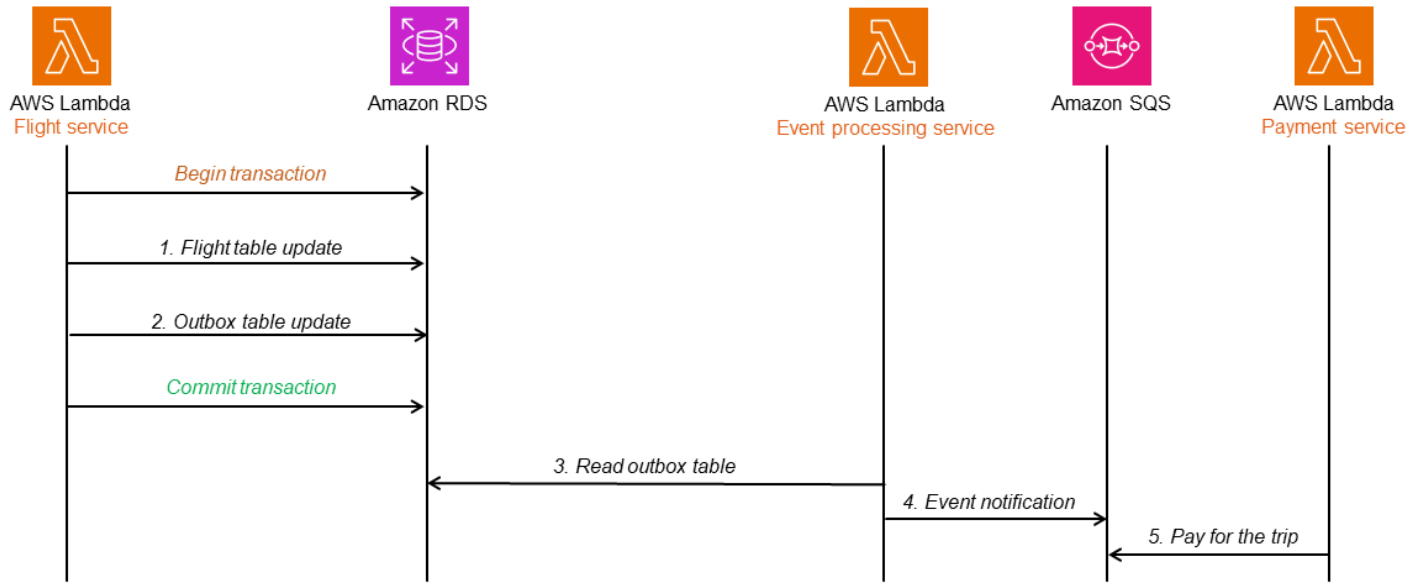
Untuk mengatasi masalah ini, Anda dapat menggunakan tabel kotak keluar atau mengubah pengambilan data (CDC). Bagian berikut membahas dua opsi ini dan bagaimana Anda dapat menerapkannya dengan menggunakan layanan AWS.

Menggunakan tabel kotak keluar dengan database relasional

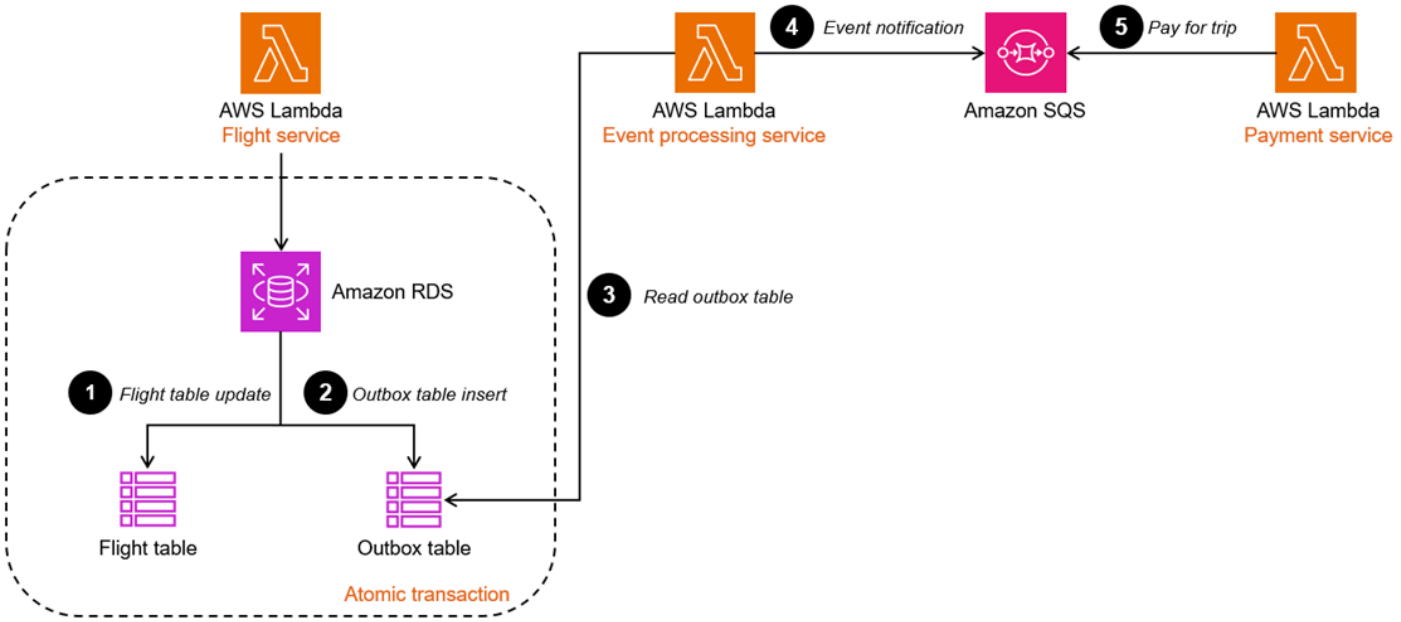
Meja kotak keluar menyimpan semua acara dari layanan penerbangan dengan stempel waktu dan nomor urut.

Ketika tabel penerbangan diperbarui, tabel kotak keluar juga diperbarui dalam transaksi yang sama. Layanan lain (misalnya, layanan pemrosesan peristiwa) membaca dari tabel kotak keluar dan mengirimkan acara ke Amazon SQS. Amazon SQS mengirimkan pesan tentang acara tersebut ke layanan pembayaran untuk diproses lebih lanjut. Antrian standar Amazon SQS menjamin bahwa pesan dikirimkan setidaknya sekali dan tidak hilang. Namun, ketika Anda menggunakan antrian standar Amazon SQS, pesan atau peristiwa yang sama mungkin dikirimkan lebih dari satu kali, jadi Anda harus memastikan bahwa layanan pemberitahuan acara idempoten (yaitu, memproses pesan yang sama beberapa kali seharusnya tidak memiliki efek buruk). Jika Anda mengharuskan pesan dikirimkan tepat sekali, dengan pemesanan pesan, Anda dapat menggunakan antrian Amazon SQS terlebih dahulu masuk, keluar pertama (FIFO).

Jika pembaruan tabel penerbangan gagal atau pembaruan tabel kotak keluar gagal, seluruh transaksi dibatalkan, sehingga tidak ada inkonsistensi data hilir.



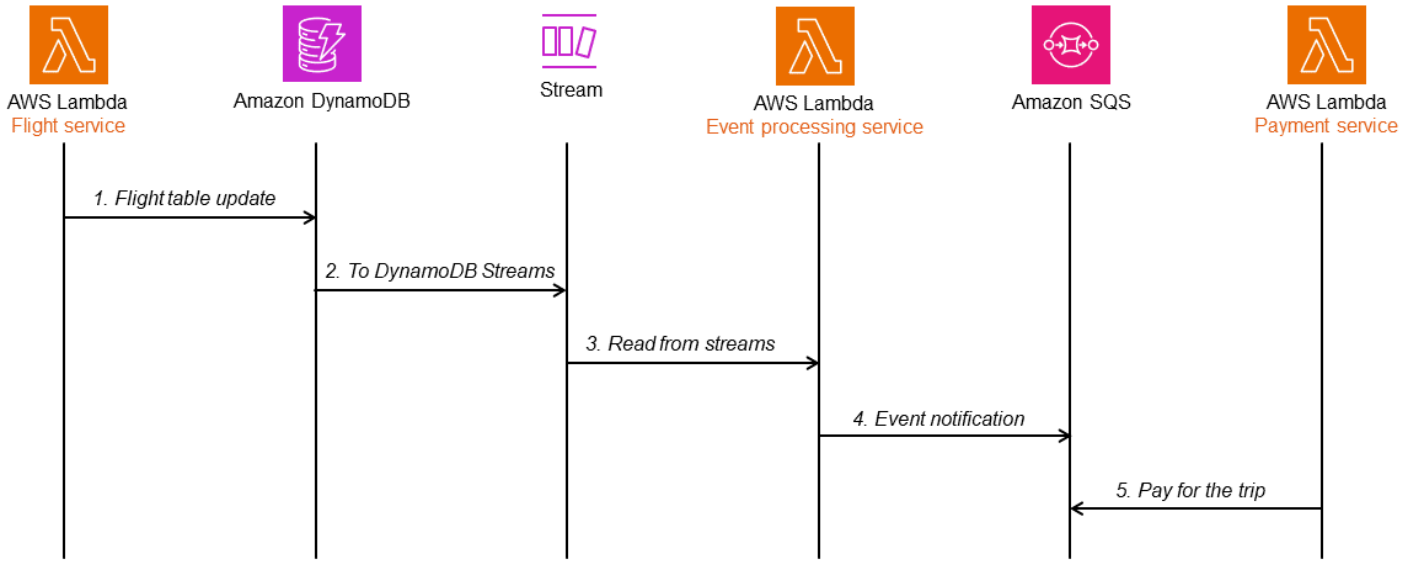
Dalam diagram berikut, arsitektur outbox transaksional diimplementasikan dengan menggunakan database Amazon RDS. Ketika layanan pemrosesan peristiwa membaca tabel kotak keluar, ia hanya mengenali baris yang merupakan bagian dari transaksi yang berkomitmen (berhasil), dan kemudian menempatkan pesan untuk acara tersebut dalam antrian SQS, yang dibaca oleh layanan pembayaran untuk diproses lebih lanjut. Desain ini menyelesaikan masalah operasi penulisan ganda dan mempertahankan urutan pesan dan peristiwa dengan menggunakan stempel waktu dan nomor urut.



Menggunakan pengambilan data perubahan (CDC)

Beberapa database mendukung penerbitan modifikasi tingkat item untuk menangkap data yang diubah. Anda dapat mengidentifikasi item yang diubah dan mengirim pemberitahuan acara yang sesuai. Ini menghemat biaya pembuatan tabel lain untuk melacak pembaruan. Acara yang diprakarsai oleh layanan penerbangan disimpan dalam atribut lain dari item yang sama.

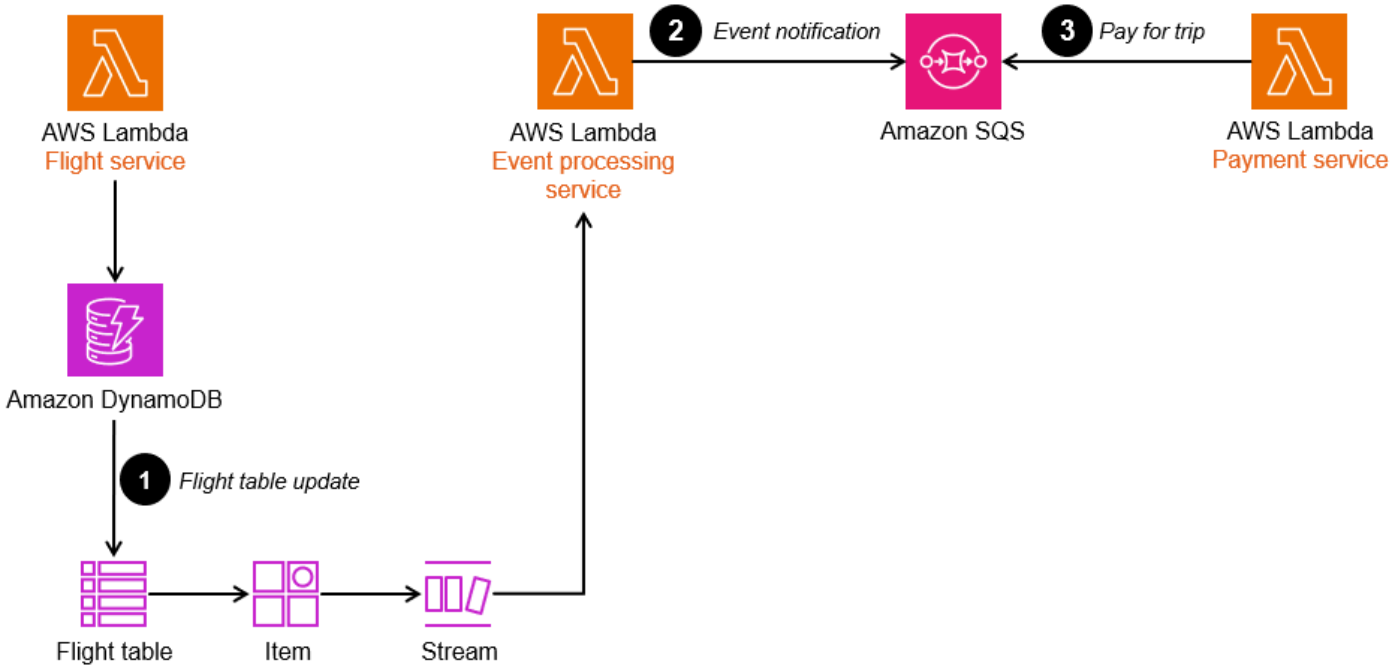
Amazon DynamoDB adalah database NoSQL nilai kunci yang mendukung pembaruan CDC. Dalam diagram urutan berikut, DynamoDB menerbitkan modifikasi tingkat item ke Amazon DynamoDB Streams. Layanan pemrosesan acara membaca dari aliran dan menerbitkan pemberitahuan acara ke layanan pembayaran untuk diproses lebih lanjut.



DynamoDB Streams menangkap aliran informasi yang berkaitan dengan perubahan tingkat item dalam tabel DynamoDB dengan menggunakan urutan urutan waktu.

Anda dapat menerapkan pola outbox transaksional dengan mengaktifkan aliran pada tabel DynamoDB. Fungsi Lambda untuk layanan pemrosesan acara dikaitkan dengan aliran ini.

- Saat tabel penerbangan diperbarui, data yang diubah ditangkap oleh DynamoDB Streams, dan layanan pemrosesan peristiwa melakukan polling aliran untuk catatan baru.
- Ketika rekaman aliran baru tersedia, fungsi Lambda secara sinkron menempatkan pesan untuk acara dalam antrean SQS untuk diproses lebih lanjut. Anda dapat menambahkan atribut ke item DynamoDB untuk menangkap stempel waktu dan nomor urut sesuai kebutuhan untuk meningkatkan kekokohan implementasi.



Kode sampel

Menggunakan tabel kotak keluar

Kode sampel di bagian ini menunjukkan bagaimana Anda dapat menerapkan pola outbox transaksional dengan menggunakan tabel outbox. Untuk melihat kode lengkap, lihat GitHubrepositori untuk contoh ini.

Cuplikan kode berikut menyimpan `Flight` entitas dan `Flight` peristiwa dalam database dalam tabel masing-masing dalam satu transaksi.

```
@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;
    }
```

Layanan terpisah bertugas memindai tabel kotak keluar secara teratur untuk acara baru, mengirimkannya ke Amazon SQS, dan menghapusnya dari tabel jika Amazon SQS berhasil merespons. Tingkat polling dapat dikonfigurasi dalam file. `application.properties`

```
@Scheduled(fixedDelayString = "${sqs.polling_ms}")
    public void forwardEventsToSQS() {
        List<FlightOutbox> entities =
 outboxRepository.findAllByOrderByIdAsc(Pageable.ofSize(batchSize)).toList();
        if (!entities.isEmpty()) {
            GetQueueUrlRequest getQueueRequest = GetQueueUrlRequest.builder()
                    .queueName(sqsQueueName)
                    .build();
            String queueUrl = this.sqsClient.getQueueUrl(getQueueRequest).queueUrl();
            List<SendMessageBatchRequestEntry> messageEntries = new ArrayList<>();
            entities.forEach(entity ->
 messageEntries.add(SendMessageBatchRequestEntry.builder()
                    .id(entity.getId().toString())
                    .messageGroupId(entity.getAggregateId())
                    .messageDeduplicationId(entity.getId().toString())
                    .messageBody(entity.getPayload().toString())
                    .build())
            );
            SendMessageBatchRequest sendMessageBatchRequest =
 SendMessageBatchRequest.builder()
                    .queueUrl(queueUrl)
                    .entries(messageEntries)
                    .build();
            sqsClient.sendMessageBatch(sendMessageBatchRequest);
            outboxRepository.deleteAllInBatch(entities);
        }
    }
```

Menggunakan pengambilan data perubahan (CDC)

Kode sampel di bagian ini menunjukkan bagaimana Anda dapat menerapkan pola outbox transaksional dengan menggunakan kapabilitas change data capture (CDC) DynamoDB. Untuk melihat kode lengkap, lihat GitHubrepositori untuk contoh ini.

Cuplikan AWS Cloud Development Kit (AWS CDK) kode berikut membuat tabel penerbangan DynamoDB dan aliran data Amazon Kinesis (`cdcStream`), dan mengonfigurasi tabel penerbangan untuk mengirim semua pembaruannya ke aliran.

```
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,
    }

});
```

Cuplikan kode dan konfigurasi berikut menentukan fungsi aliran awan musim semi yang mengambil pembaruan dalam aliran Kinesis dan meneruskan peristiwa ini ke antrian SQS untuk diproses lebih lanjut.

```
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 repositori

Untuk implementasi lengkap arsitektur sampel untuk pola ini, lihat GitHub repositori di. https://github.com/aws-samples/transactional-outbox-pattern

Sumber daya

Referensi

- AWS Pusat Arsitektur
- AWS Pusat Pengembang
- Perpustakaan Pembangun Amazon

Alat

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

Metodologi

- Aplikasi Dua Belas Faktor (ePub oleh Adam Wiggins)
- Nygard, Michael T. Lepaskan! : Merancang dan Menyebarkan Perangkat Lunak Siap Produksi. Edisi ke-2. Raleigh, NC: Rak Buku Pragmatis, 2018.
- Kegigihan Polyglot (posting blog oleh Martin Fowler)
- StranglerFigApplication(posting blog oleh Martin Fowler)

Riwayat dokumen

Tabel berikut menjelaskan perubahan signifikan pada panduan ini. Jika Anda ingin diberi tahu tentang pembaruan masa depan, Anda dapat berlangganan umpan RSS.

Perubahan	Deskripsi	Tanggal
Pola baru	Menambahkan dua pola baru: arsitektur heksagonal dan scatter-gathering.	7 Mei 2024
Contoh kode baru	Menambahkan kode sampel untuk kasus penggunaan change data capture (CDC) ke pola pola outbox transaksi onal.	Februari 23, 2024
Contoh kode baru	• Memperbarui pola outbox transaksional dengan kode sampel. • Menghapus bagian tentang pola orkestrasi dan koreografi, yang digantika n oleh koreografi saga dan orkestrasi saga.	16 November 2023
Pola baru	Menambahkan tiga pola baru: koreografi saga, publish-s ubscribe, dan sumber acara.	14 November 2023
Perbarui	Memperbarui bagian implementasi pola ara strangler.	2 Oktober 2023
Publikasi awal	Rilis pertama ini mencakup delapan pola desain: lapisan	28 Juli 2023

anti-korupsi (ACL), perutean API, pemutus sirkuit, orkestrasi dan koreografi, coba lagi dengan backoff, orkestrasi saga, ara pencekik, dan kotak keluar transaksional.

AWS Glosarium Panduan Preskriptif

Berikut ini adalah istilah yang umum digunakan dalam strategi, panduan, dan pola yang disediakan oleh Panduan AWS Preskriptif. Untuk menyarankan entri, silakan gunakan tautan Berikan umpan balik di akhir glosarium.

Nomor

7 Rs

Tujuh strategi migrasi umum untuk memindahkan aplikasi ke cloud. Strategi ini dibangun di atas 5 Rs yang diidentifikasi Gartner pada tahun 2011 dan terdiri dari yang berikut:

- Refactor/Re-Architect — Memindahkan aplikasi dan memodifikasi arsitekturnya dengan memanfaatkan sepenuhnya fitur cloud-native untuk meningkatkan kelincahan, kinerja, dan skalabilitas. Ini biasanya melibatkan porting sistem operasi dan database. Contoh: Migrasikan database Oracle lokal Anda ke Amazon Aurora PostgreSQL Compatible Edition.
- Replatform (angkat dan bentuk ulang) — Pindahkan aplikasi ke cloud, dan perkenalkan beberapa tingkat pengoptimalan untuk memanfaatkan kemampuan cloud. Contoh: Memigrasikan database Oracle lokal Anda ke Amazon Relational Database Service (Amazon RDS) untuk Oracle di. AWS Cloud
- Pembelian kembali (drop and shop) - Beralih ke produk yang berbeda, biasanya dengan beralih dari lisensi tradisional ke model SaaS. Contoh: Migrasikan sistem manajemen hubungan pelanggan (CRM) Anda ke Salesforce.com.
- Rehost (lift dan shift) — Pindahkan aplikasi ke cloud tanpa membuat perubahan apa pun untuk memanfaatkan kemampuan cloud. Contoh: Migrasikan database Oracle lokal Anda ke Oracle pada instance EC2 di. AWS Cloud
- Relokasi (hypervisor-level lift and shift) — Pindahkan infrastruktur ke cloud tanpa membeli perangkat keras baru, menulis ulang aplikasi, atau memodifikasi operasi yang ada. Anda memigrasikan server dari platform lokal ke layanan cloud untuk platform yang sama. Contoh: Migrasikan Microsoft Hyper-V aplikasi ke AWS.
- Pertahankan (kunjungi kembali) - Simpan aplikasi di lingkungan sumber Anda. Ini mungkin termasuk aplikasi yang memerlukan refactoring besar, dan Anda ingin menunda pekerjaan itu sampai nanti, dan aplikasi lama yang ingin Anda pertahankan, karena tidak ada pembenaran bisnis untuk memigrasikannya.

- Pensiun — Menonaktifkan atau menghapus aplikasi yang tidak lagi diperlukan di lingkungan sumber Anda.

A

ABAC

Lihat kontrol akses berbasis atribut.

layanan abstrak

Lihat layanan terkelola.

ASAM

Lihat atomisitas, konsistensi, isolasi, daya tahan.

migrasi aktif-aktif

Metode migrasi database di mana database sumber dan target tetap sinkron (dengan menggunakan alat replikasi dua arah atau operasi penulisan ganda), dan kedua database menangani transaksi dari menghubungkan aplikasi selama migrasi. Metode ini mendukung migrasi dalam batch kecil yang terkontrol alih-alih memerlukan pemotongan satu kali. Ini lebih fleksibel tetapi membutuhkan lebih banyak pekerjaan daripada migrasi aktif-pasif.

migrasi aktif-pasif

Metode migrasi database di mana database sumber dan target disimpan dalam sinkron, tetapi hanya database sumber yang menangani transaksi dari menghubungkan aplikasi sementara data direplikasi ke database target. Basis data target tidak menerima transaksi apa pun selama migrasi.

fungsi agregat

Fungsi SQL yang beroperasi pada sekelompok baris dan menghitung nilai pengembalian tunggal untuk grup. Contoh fungsi agregat meliputi `SUM` dan`MAX`.

AI

Lihat kecerdasan buatan.

AIOps

Lihat operasi kecerdasan buatan.

anonimisasi

Proses menghapus informasi pribadi secara permanen dalam kumpulan data. Anonimisasi dapat membantu melindungi privasi pribadi. Data anonim tidak lagi dianggap sebagai data pribadi.

anti-pola

Solusi yang sering digunakan untuk masalah berulang di mana solusinya kontra-produktif, tidak efektif, atau kurang efektif daripada alternatif.

kontrol aplikasi

Pendekatan keamanan yang memungkinkan penggunaan hanya aplikasi yang disetujui untuk membantu melindungi sistem dari malware.

portofolio aplikasi

Kumpulan informasi rinci tentang setiap aplikasi yang digunakan oleh organisasi, termasuk biaya untuk membangun dan memelihara aplikasi, dan nilai bisnisnya. Informasi ini adalah kunci untuk penemuan portofolio dan proses analisis dan membantu mengidentifikasi dan memprioritaskan aplikasi yang akan dimigrasi, dimodernisasi, dan dioptimalkan.

kecerdasan buatan (AI)

Bidang ilmu komputer yang didedikasikan untuk menggunakan teknologi komputasi untuk melakukan fungsi kognitif yang biasanya terkait dengan manusia, seperti belajar, memecahkan masalah, dan mengenali pola. Untuk informasi lebih lanjut, lihat Apa itu Kecerdasan Buatan?

operasi kecerdasan buatan (AIOps)

Proses menggunakan teknik pembelajaran mesin untuk memecahkan masalah operasional, mengurangi insiden operasional dan intervensi manusia, dan meningkatkan kualitas layanan. Untuk informasi selengkapnya tentang cara AIOps digunakan dalam strategi AWS migrasi, lihat panduan integrasi operasi.

enkripsi asimetris

Algoritma enkripsi yang menggunakan sepasang kunci, kunci publik untuk enkripsi dan kunci pribadi untuk dekripsi. Anda dapat berbagi kunci publik karena tidak digunakan untuk dekripsi, tetapi akses ke kunci pribadi harus sangat dibatasi.

atomisitas, konsistensi, isolasi, daya tahan (ACID)

Satu set properti perangkat lunak yang menjamin validitas data dan keandalan operasional database, bahkan dalam kasus kesalahan, kegagalan daya, atau masalah lainnya.

kontrol akses berbasis atribut (ABAC)

Praktik membuat izin berbutir halus berdasarkan atribut pengguna, seperti departemen, peran pekerjaan, dan nama tim. Untuk informasi selengkapnya, lihat ABAC untuk AWS dokumentasi AWS Identity and Access Management (IAM).

sumber data otoritatif

Lokasi di mana Anda menyimpan versi utama data, yang dianggap sebagai sumber informasi yang paling dapat diandalkan. Anda dapat menyalin data dari sumber data otoritatif ke lokasi lain untuk tujuan memproses atau memodifikasi data, seperti menganonimkan, menyunting, atau membuat nama samaran.

Zona Ketersediaan

Lokasi berbeda di dalam Wilayah AWS yang terisolasi dari kegagalan di Availability Zone lainnya dan menyediakan konektivitas jaringan latensi rendah yang murah ke Availability Zone lainnya di Wilayah yang sama.

AWS Kerangka Adopsi Cloud (AWS CAF)

Kerangka pedoman dan praktik terbaik AWS untuk membantu organisasi mengembangkan rencana yang efisien dan efektif untuk bergerak dengan sukses ke cloud. AWS CAF mengatur panduan ke dalam enam area fokus yang disebut perspektif: bisnis, orang, tata kelola, platform, keamanan, dan operasi. Perspektif bisnis, orang, dan tata kelola fokus pada keterampilan dan proses bisnis; perspektif platform, keamanan, dan operasi fokus pada keterampilan dan proses teknis. Misalnya, perspektif masyarakat menargetkan pemangku kepentingan yang menangani sumber daya manusia (SDM), fungsi kepegawaian, dan manajemen orang. Untuk perspektif ini, AWS CAF memberikan panduan untuk pengembangan, pelatihan, dan komunikasi orang untuk membantu mempersiapkan organisasi untuk adopsi cloud yang sukses. Untuk informasi lebih lanjut, lihat situs web AWS CAF dan whitepaper AWS CAF.

AWS Kerangka Kualifikasi Beban Kerja (AWS WQF)

Alat yang mengevaluasi beban kerja migrasi database, merekomendasikan strategi migrasi, dan memberikan perkiraan kerja. AWS WQF disertakan dengan AWS Schema Conversion Tool ()AWS SCT. Ini menganalisis skema database dan objek kode, kode aplikasi, dependensi, dan karakteristik kinerja, dan memberikan laporan penilaian.

B

bot buruk

Bot yang dimaksudkan untuk mengganggu atau membahayakan individu atau organisasi.

BCP

Lihat perencanaan kontinuitas bisnis.

grafik perilaku

Pandangan interaktif yang terpadu tentang perilaku dan interaksi sumber daya dari waktu ke waktu. Anda dapat menggunakan grafik perilaku dengan Amazon Detective untuk memeriksa upaya logon yang gagal, panggilan API yang mencurigakan, dan tindakan serupa. Untuk informasi selengkapnya, lihat Data dalam grafik perilaku di dokumentasi Detektif.

sistem big-endian

Sistem yang menyimpan byte paling signifikan terlebih dahulu. Lihat juga endianness.

klasifikasi biner

Sebuah proses yang memprediksi hasil biner (salah satu dari dua kelas yang mungkin). Misalnya, model ML Anda mungkin perlu memprediksi masalah seperti “Apakah email ini spam atau bukan spam?” atau “Apakah produk ini buku atau mobil?”

filter mekar

Struktur data probabilistik dan efisien memori yang digunakan untuk menguji apakah suatu elemen adalah anggota dari suatu himpunan.

deployment biru/hijau

Strategi penyebaran tempat Anda membuat dua lingkungan yang terpisah namun identik. Anda menjalankan versi aplikasi saat ini di satu lingkungan (biru) dan versi aplikasi baru di lingkungan lain (hijau). Strategi ini membantu Anda dengan cepat memutar kembali dengan dampak minimal.

bot

Aplikasi perangkat lunak yang menjalankan tugas otomatis melalui internet dan mensimulasikan aktivitas atau interaksi manusia. Beberapa bot berguna atau bermanfaat, seperti perayap web yang mengindeks informasi di internet. Beberapa bot lain, yang dikenal sebagai bot buruk, dimaksudkan untuk mengganggu atau membahayakan individu atau organisasi.

botnet

Jaringan bot yang terinfeksi oleh malware dan berada di bawah kendali satu pihak, yang dikenal sebagai bot herder atau operator bot. Botnet adalah mekanisme paling terkenal untuk skala bot dan dampaknya.

cabang

Area berisi repositori kode. Cabang pertama yang dibuat dalam repositori adalah cabang utama. Anda dapat membuat cabang baru dari cabang yang ada, dan Anda kemudian dapat mengembangkan fitur atau memperbaiki bug di cabang baru. Cabang yang Anda buat untuk membangun fitur biasanya disebut sebagai cabang fitur. Saat fitur siap dirilis, Anda menggabungkan cabang fitur kembali ke cabang utama. Untuk informasi selengkapnya, lihat Tentang cabang (GitHub dokumentasi).

akses break-glass

Dalam keadaan luar biasa dan melalui proses yang disetujui, cara cepat bagi pengguna untuk mendapatkan akses ke Akun AWS yang biasanya tidak memiliki izin untuk mengaksesnya. Untuk informasi lebih lanjut, lihat indikator Implementasikan prosedur break-glass dalam panduan Well-Architected AWS .

strategi brownfield

Infrastruktur yang ada di lingkungan Anda. Saat mengadopsi strategi brownfield untuk arsitektur sistem, Anda merancang arsitektur di sekitar kendala sistem dan infrastruktur saat ini. Jika Anda memperluas infrastruktur yang ada, Anda dapat memadukan strategi brownfield dan greenfield.

cache penyangga

Area memori tempat data yang paling sering diakses disimpan.

kemampuan bisnis

Apa yang dilakukan bisnis untuk menghasilkan nilai (misalnya, penjualan, layanan pelanggan, atau pemasaran). Arsitektur layanan mikro dan keputusan pengembangan dapat didorong oleh kemampuan bisnis. Untuk informasi selengkapnya, lihat bagian Terorganisir di sekitar kemampuan bisnis dari Menjalankan layanan mikro kontainer di whitepaper. AWS

perencanaan kelangsungan bisnis (BCP)

Rencana yang membahas dampak potensial dari peristiwa yang mengganggu, seperti migrasi skala besar, pada operasi dan memungkinkan bisnis untuk melanjutkan operasi dengan cepat.

C

KAFE

Lihat Kerangka Adopsi AWS Cloud.

penyebaran kenari

Rilis versi yang lambat dan bertahap untuk pengguna akhir. Ketika Anda yakin, Anda menyebarkan versi baru dan mengganti versi saat ini secara keseluruhan.

CCoE

Lihat Cloud Center of Excellence.

CDC

Lihat mengubah pengambilan data.

ubah pengambilan data (CDC)

Proses melacak perubahan ke sumber data, seperti tabel database, dan merekam metadata tentang perubahan tersebut. Anda dapat menggunakan CDC untuk berbagai tujuan, seperti mengaudit atau mereplikasi perubahan dalam sistem target untuk mempertahankan sinkronisasi.

rekayasa kekacauan

Dengan sengaja memperkenalkan kegagalan atau peristiwa yang mengganggu untuk menguji ketahanan sistem. Anda dapat menggunakan AWS Fault Injection Service (AWS FIS) untuk melakukan eksperimen yang menekankan AWS beban kerja Anda dan mengevaluasi responsnya.

CI/CD

Lihat integrasi berkelanjutan dan pengiriman berkelanjutan.

klasifikasi

Proses kategorisasi yang membantu menghasilkan prediksi. Model ML untuk masalah klasifikasi memprediksi nilai diskrit. Nilai diskrit selalu berbeda satu sama lain. Misalnya, model mungkin perlu mengevaluasi apakah ada mobil dalam gambar atau tidak.

Enkripsi sisi klien

Enkripsi data secara lokal, sebelum target Layanan AWS menerimanya.

Pusat Keunggulan Cloud (CCoE)

Tim multi-disiplin yang mendorong upaya adopsi cloud di seluruh organisasi, termasuk mengembangkan praktik terbaik cloud, memobilisasi sumber daya, menetapkan jadwal migrasi, dan memimpin organisasi melalui transformasi skala besar. Untuk informasi selengkapnya, lihat posting CCo E di Blog Strategi AWS Cloud Perusahaan.

komputasi cloud

Teknologi cloud yang biasanya digunakan untuk penyimpanan data jarak jauh dan manajemen perangkat IoT. Cloud computing umumnya terhubung ke teknologi edge computing.

model operasi cloud

Dalam organisasi TI, model operasi yang digunakan untuk membangun, mematangkan, dan mengoptimalkan satu atau lebih lingkungan cloud. Untuk informasi selengkapnya, lihat Membangun Model Operasi Cloud Anda.

tahap adopsi cloud

Empat fase yang biasanya dilalui organisasi ketika mereka bermigrasi ke AWS Cloud:

- Proyek — Menjalankan beberapa proyek terkait cloud untuk bukti konsep dan tujuan pembelajaran
- Foundation — Melakukan investasi dasar untuk meningkatkan adopsi cloud Anda (misalnya, membuat landing zone, mendefinisikan CCo E, membuat model operasi)
- Migrasi — Migrasi aplikasi individual
- Re-invention — Mengoptimalkan produk dan layanan, dan berinovasi di cloud

Tahapan ini didefinisikan oleh Stephen Orban dalam posting blog The Journey Toward Cloud-First & the Stages of Adoption di blog Strategi Perusahaan. AWS Cloud Untuk informasi tentang bagaimana kaitannya dengan strategi AWS migrasi, lihat panduan kesiapan migrasi.

CMDB

Lihat database manajemen konfigurasi.

repositori kode

Lokasi di mana kode sumber dan aset lainnya, seperti dokumentasi, sampel, dan skrip, disimpan dan diperbarui melalui proses kontrol versi. Repositori cloud umum termasuk GitHub atau. Bitbucket Cloud Setiap versi kode disebut cabang. Dalam struktur layanan mikro, setiap repositori

dikhususkan untuk satu bagian fungsionalitas. Pipa CI/CD tunggal dapat menggunakan beberapa repositori.

cache dingin

Cache buffer yang kosong, tidak terisi dengan baik, atau berisi data basi atau tidak relevan. Ini mempengaruhi kinerja karena instance database harus membaca dari memori utama atau disk, yang lebih lambat daripada membaca dari cache buffer.

data dingin

Data yang jarang diakses dan biasanya historis. Saat menanyakan jenis data ini, kueri lambat biasanya dapat diterima. Memindahkan data ini ke tingkat atau kelas penyimpanan yang berkinerja lebih rendah dan lebih murah dapat mengurangi biaya.

visi komputer (CV)

Bidang AI yang menggunakan pembelajaran mesin untuk menganalisis dan mengekstrak informasi dari format visual seperti gambar dan video digital. Misalnya, Amazon SageMaker AI menyediakan algoritma pemrosesan gambar untuk CV.

konfigurasi drift

Untuk beban kerja, konfigurasi berubah dari status yang diharapkan. Ini dapat menyebabkan beban kerja menjadi tidak patuh, dan biasanya bertahap dan tidak disengaja.

database manajemen konfigurasi (CMDB)

Repositori yang menyimpan dan mengelola informasi tentang database dan lingkungan TI, termasuk komponen perangkat keras dan perangkat lunak dan konfigurasinya. Anda biasanya menggunakan data dari CMDB dalam penemuan portofolio dan tahap analisis migrasi.

paket kesesuaian

Kumpulan AWS Config aturan dan tindakan remediasi yang dapat Anda kumpulkan untuk menyesuaikan kepatuhan dan pemeriksaan keamanan Anda. Anda dapat menerapkan paket kesesuaian sebagai entitas tunggal di Akun AWS dan Region, atau di seluruh organisasi, dengan menggunakan templat YAMM. Untuk informasi selengkapnya, lihat Paket kesesuaian dalam dokumentasi. AWS Config

integrasi berkelanjutan dan pengiriman berkelanjutan (CI/CD)

Proses mengotomatiskan sumber, membangun, menguji, pementasan, dan tahap produksi dari proses rilis perangkat lunak. CI/CD is commonly described as a pipeline. CI/CDdapat membantu

Anda mengotomatiskan proses, meningkatkan produktivitas, meningkatkan kualitas kode, dan memberikan lebih cepat. Untuk informasi lebih lanjut, lihat Manfaat pengiriman berkelanjutan. CD juga dapat berarti penerapan berkelanjutan. Untuk informasi selengkapnya, lihat Continuous Delivery vs Continuous Deployment.

CV

Lihat visi komputer.

D

data saat istirahat

Data yang stasioner di jaringan Anda, seperti data yang ada di penyimpanan.

klasifikasi data

Proses untuk mengidentifikasi dan mengkategorikan data dalam jaringan Anda berdasarkan kekritisan dan sensitivitasnya. Ini adalah komponen penting dari setiap strategi manajemen risiko keamanan siber karena membantu Anda menentukan perlindungan dan kontrol retensi yang tepat untuk data. Klasifikasi data adalah komponen pilar keamanan dalam AWS Well-Architected Framework. Untuk informasi selengkapnya, lihat Klasifikasi data.

penyimpangan data

Variasi yang berarti antara data produksi dan data yang digunakan untuk melatih model ML, atau perubahan yang berarti dalam data input dari waktu ke waktu. Penyimpangan data dapat mengurangi kualitas, akurasi, dan keadilan keseluruhan dalam prediksi model ML.

data dalam transit

Data yang aktif bergerak melalui jaringan Anda, seperti antara sumber daya jaringan.

jala data

Kerangka arsitektur yang menyediakan kepemilikan data terdistribusi dan terdesentralisasi dengan manajemen dan tata kelola terpusat.

minimalisasi data

Prinsip pengumpulan dan pemrosesan hanya data yang sangat diperlukan. Mempraktikkan minimalisasi data di dalamnya AWS Cloud dapat mengurangi risiko privasi, biaya, dan jejak karbon analitik Anda.

perimeter data

Satu set pagar pembatas pencegahan di AWS lingkungan Anda yang membantu memastikan bahwa hanya identitas tepercaya yang mengakses sumber daya tepercaya dari jaringan yang diharapkan. Untuk informasi selengkapnya, lihat Membangun perimeter data pada AWS.

prapemrosesan data

Untuk mengubah data mentah menjadi format yang mudah diuraikan oleh model ML Anda. Preprocessing data dapat berarti menghapus kolom atau baris tertentu dan menangani nilai yang hilang, tidak konsisten, atau duplikat.

asal data

Proses melacak asal dan riwayat data sepanjang siklus hidupnya, seperti bagaimana data dihasilkan, ditransmisikan, dan disimpan.

subjek data

Individu yang datanya dikumpulkan dan diproses.

gudang data

Sistem manajemen data yang mendukung intelijen bisnis, seperti analitik. Gudang data biasanya berisi sejumlah besar data historis, dan biasanya digunakan untuk kueri dan analisis.

bahasa definisi database (DDL)

Pernyataan atau perintah untuk membuat atau memodifikasi struktur tabel dan objek dalam database.

bahasa manipulasi basis data (DHTML)

Pernyataan atau perintah untuk memodifikasi (memasukkan, memperbarui, dan menghapus) informasi dalam database.

DDL

Lihat bahasa definisi database.

ansambel yang dalam

Untuk menggabungkan beberapa model pembelajaran mendalam untuk prediksi. Anda dapat menggunakan ansambel dalam untuk mendapatkan prediksi yang lebih akurat atau untuk memperkirakan ketidakpastian dalam prediksi.

pembelajaran mendalam

Subbidang ML yang menggunakan beberapa lapisan jaringan saraf tiruan untuk mengidentifikasi pemetaan antara data input dan variabel target yang diinginkan.

defense-in-depth

Pendekatan keamanan informasi di mana serangkaian mekanisme dan kontrol keamanan dilapisi dengan cermat di seluruh jaringan komputer untuk melindungi kerahasiaan, integritas, dan ketersediaan jaringan dan data di dalamnya. Saat Anda mengadopsi strategi ini AWS, Anda menambahkan beberapa kontrol pada lapisan AWS Organizations struktur yang berbeda untuk membantu mengamankan sumber daya. Misalnya, defense-in-depth pendekatan mungkin menggabungkan otentikasi multi-faktor, segmentasi jaringan, dan enkripsi.

administrator yang didelegasikan

Di AWS Organizations, layanan yang kompatibel dapat mendaftarkan akun AWS anggota untuk mengelola akun organisasi dan mengelola izin untuk layanan tersebut. Akun ini disebut administrator yang didelegasikan untuk layanan itu. Untuk informasi selengkapnya dan daftar layanan yang kompatibel, lihat Layanan yang berfungsi dengan AWS Organizations AWS Organizations dokumentasi.

deployment

Proses pembuatan aplikasi, fitur baru, atau perbaikan kode tersedia di lingkungan target. Deployment melibatkan penerapan perubahan dalam basis kode dan kemudian membangun dan menjalankan basis kode itu di lingkungan aplikasi.

lingkungan pengembangan

Lihat lingkungan.

kontrol detektif

Kontrol keamanan yang dirancang untuk mendeteksi, mencatat, dan memperingatkan setelah suatu peristiwa terjadi. Kontrol ini adalah garis pertahanan kedua, memperingatkan Anda tentang peristiwa keamanan yang melewati kontrol pencegahan di tempat. Untuk informasi selengkapnya, lihat Kontrol Detektif dalam Menerapkan kontrol keamanan pada. AWS

pemetaan aliran nilai pengembangan (DVSM)

Sebuah proses yang digunakan untuk mengidentifikasi dan memprioritaskan kendala yang mempengaruhi kecepatan dan kualitas dalam siklus hidup pengembangan perangkat lunak. DVSM memperluas proses pemetaan aliran nilai yang awalnya dirancang untuk praktik

manufaktur ramping. Ini berfokus pada langkah-langkah dan tim yang diperlukan untuk menciptakan dan memindahkan nilai melalui proses pengembangan perangkat lunak.

kembar digital

Representasi virtual dari sistem dunia nyata, seperti bangunan, pabrik, peralatan industri, atau jalur produksi. Kembar digital mendukung pemeliharaan prediktif, pemantauan jarak jauh, dan optimalisasi produksi.

tabel dimensi

Dalam skema bintang, tabel yang lebih kecil yang berisi atribut data tentang data kuantitatif dalam tabel fakta. Atribut tabel dimensi biasanya bidang teks atau angka diskrit yang berperilaku seperti teks. Atribut ini biasanya digunakan untuk pembatasan kueri, pemfilteran, dan pelabelan set hasil.

musibah

Peristiwa yang mencegah beban kerja atau sistem memenuhi tujuan bisnisnya di lokasi utama yang digunakan. Peristiwa ini dapat berupa bencana alam, kegagalan teknis, atau akibat dari tindakan manusia, seperti kesalahan konfigurasi yang tidak disengaja atau serangan malware.

pemulihan bencana (DR)

Strategi dan proses yang Anda gunakan untuk meminimalkan downtime dan kehilangan data yang disebabkan oleh bencana. Untuk informasi selengkapnya, lihat Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML~

Lihat bahasa manipulasi basis data.

desain berbasis domain

Pendekatan untuk mengembangkan sistem perangkat lunak yang kompleks dengan menghubungkan komponennya ke domain yang berkembang, atau tujuan bisnis inti, yang dilayani oleh setiap komponen. Konsep ini diperkenalkan oleh Eric Evans dalam bukunya, Domain-Driven Design: Tackling Complexity in the Heart of Software (Boston: Addison-Wesley Professional, 2003). Untuk informasi tentang cara menggunakan desain berbasis domain dengan pola gambar pencekik, lihat Memodernisasi layanan web Microsoft ASP.NET (ASMX) lama secara bertahap menggunakan container dan Amazon API Gateway.

DR

Lihat pemulihan bencana.

deteksi drift

Melacak penyimpangan dari konfigurasi dasar. Misalnya, Anda dapat menggunakan AWS CloudFormation untuk mendeteksi penyimpangan dalam sumber daya sistem, atau Anda dapat menggunakannya AWS Control Tower untuk mendeteksi perubahan di landing zone yang mungkin memengaruhi kepatuhan terhadap persyaratan tata kelola.

DVSM

Lihat pemetaan aliran nilai pengembangan.

E

EDA

Lihat analisis data eksplorasi.

EDI

Lihat pertukaran data elektronik.

komputasi tepi

Teknologi yang meningkatkan daya komputasi untuk perangkat pintar di tepi jaringan IoT. Jika dibandingkan dengan komputasi awan, komputasi tepi dapat mengurangi latensi komunikasi dan meningkatkan waktu respons.

pertukaran data elektronik (EDI)

Pertukaran otomatis dokumen bisnis antar organisasi. Untuk informasi selengkapnya, lihat Apa itu Pertukaran Data Elektronik.

enkripsi

Proses komputasi yang mengubah data plaintext, yang dapat dibaca manusia, menjadi ciphertext.

kunci enkripsi

String kriptografi dari bit acak yang dihasilkan oleh algoritma enkripsi. Panjang kunci dapat bervariasi, dan setiap kunci dirancang agar tidak dapat diprediksi dan unik.

endianness

Urutan byte disimpan dalam memori komputer. Sistem big-endian menyimpan byte paling signifikan terlebih dahulu. Sistem little-endian menyimpan byte paling tidak signifikan terlebih dahulu.

titik akhir

Lihat titik akhir layanan.

layanan endpoint

Layanan yang dapat Anda host di cloud pribadi virtual (VPC) untuk dibagikan dengan pengguna lain. Anda dapat membuat layanan endpoint dengan AWS PrivateLink dan memberikan izin kepada prinsipal lain Akun AWS atau ke AWS Identity and Access Management (IAM). Akun atau prinsipal ini dapat terhubung ke layanan endpoint Anda secara pribadi dengan membuat titik akhir VPC antarmuka. Untuk informasi selengkapnya, lihat Membuat layanan titik akhir di dokumentasi Amazon Virtual Private Cloud (Amazon VPC).

perencanaan sumber daya perusahaan (ERP)

Sistem yang mengotomatiskan dan mengelola proses bisnis utama (seperti akuntansi, MES, dan manajemen proyek) untuk suatu perusahaan.

enkripsi amplop

Proses mengenkripsi kunci enkripsi dengan kunci enkripsi lain. Untuk informasi selengkapnya, lihat Enkripsi amplop dalam dokumentasi AWS Key Management Service (AWS KMS).

lingkungan

Sebuah contoh dari aplikasi yang sedang berjalan. Berikut ini adalah jenis lingkungan yang umum dalam komputasi awan:

- Development Environment — Sebuah contoh dari aplikasi yang berjalan yang hanya tersedia untuk tim inti yang bertanggung jawab untuk memelihara aplikasi. Lingkungan pengembangan digunakan untuk menguji perubahan sebelum mempromosikannya ke lingkungan atas. Jenis lingkungan ini kadang-kadang disebut sebagai lingkungan pengujian.
- lingkungan yang lebih rendah — Semua lingkungan pengembangan untuk aplikasi, seperti yang digunakan untuk build awal dan pengujian.
- lingkungan produksi — Sebuah contoh dari aplikasi yang berjalan yang pengguna akhir dapat mengakses. Dalam pipa CI/CD, lingkungan produksi adalah lingkungan penyebaran terakhir.
- lingkungan atas — Semua lingkungan yang dapat diakses oleh pengguna selain tim pengembangan inti. Ini dapat mencakup lingkungan produksi, lingkungan praproduksi, dan lingkungan untuk pengujian penerimaan pengguna.

epik

Dalam metodologi tangkas, kategori fungsional yang membantu mengatur dan memprioritaskan pekerjaan Anda. Epik memberikan deskripsi tingkat tinggi tentang persyaratan dan tugas implementasi. Misalnya, epos keamanan AWS CAF mencakup manajemen identitas dan akses, kontrol detektif, keamanan infrastruktur, perlindungan data, dan respons insiden. Untuk informasi selengkapnya tentang epos dalam strategi AWS migrasi, lihat panduan implementasi program.

ERP

Lihat perencanaan sumber daya perusahaan.

analisis data eksplorasi (EDA)

Proses menganalisis dataset untuk memahami karakteristik utamanya. Anda mengumpulkan atau mengumpulkan data dan kemudian melakukan penyelidikan awal untuk menemukan pola, mendeteksi anomali, dan memeriksa asumsi. EDA dilakukan dengan menghitung statistik ringkasan dan membuat visualisasi data.

F

tabel fakta

Tabel tengah dalam skema bintang. Ini menyimpan data kuantitatif tentang operasi bisnis. Biasanya, tabel fakta berisi dua jenis kolom: kolom yang berisi ukuran dan yang berisi kunci asing ke tabel dimensi.

gagal cepat

Filosofi yang menggunakan pengujian yang sering dan bertahap untuk mengurangi siklus hidup pengembangan. Ini adalah bagian penting dari pendekatan tangkas.

batas isolasi kesalahan

Dalam AWS Cloud, batas seperti Availability Zone, Wilayah AWS, control plane, atau data plane yang membatasi efek kegagalan dan membantu meningkatkan ketahanan beban kerja. Untuk informasi selengkapnya, lihat Batas Isolasi AWS Kesalahan.

cabang fitur

Lihat cabang.

fitur

Data input yang Anda gunakan untuk membuat prediksi. Misalnya, dalam konteks manufaktur, fitur bisa berupa gambar yang diambil secara berkala dari lini manufaktur.

pentingnya fitur

Seberapa signifikan fitur untuk prediksi model. Ini biasanya dinyatakan sebagai skor numerik yang dapat dihitung melalui berbagai teknik, seperti Shapley Additive Explanations (SHAP) dan gradien terintegrasi. Untuk informasi lebih lanjut, lihat Interpretabilitas model pembelajaran mesin dengan. AWS

transformasi fitur

Untuk mengoptimalkan data untuk proses ML, termasuk memperkaya data dengan sumber tambahan, menskalakan nilai, atau mengekstrak beberapa set informasi dari satu bidang data. Hal ini memungkinkan model ML untuk mendapatkan keuntungan dari data. Misalnya, jika Anda memecah tanggal “2021-05-27 00:15:37" menjadi “2021", “Mei”, “Kamis”, dan “15", Anda dapat membantu algoritme pembelajaran mempelajari pola bernuansa yang terkait dengan komponen data yang berbeda.

beberapa tembakan mendorong

Menyediakan LLM dengan sejumlah kecil contoh yang menunjukkan tugas dan output yang diinginkan sebelum memintanya untuk melakukan tugas serupa. Teknik ini adalah aplikasi pembelajaran dalam konteks, di mana model belajar dari contoh (bidikan) yang tertanam dalam petunjuk. Beberapa bidikan dapat efektif untuk tugas-tugas yang memerlukan pemformatan, penalaran, atau pengetahuan domain tertentu. Lihat juga bidikan nol.

FGAC

Lihat kontrol akses berbutir halus.

kontrol akses berbutir halus (FGAC)

Penggunaan beberapa kondisi untuk mengizinkan atau menolak permintaan akses.

migrasi flash-cut

Metode migrasi database yang menggunakan replikasi data berkelanjutan melalui pengambilan data perubahan untuk memigrasikan data dalam waktu sesingkat mungkin, alih-alih menggunakan pendekatan bertahap. Tujuannya adalah untuk menjaga downtime seminimal mungkin.

FM

Lihat model pondasi.

model pondasi (FM)

Jaringan saraf pembelajaran mendalam yang besar yang telah melatih kumpulan data besar-besaran data umum dan tidak berlabel. FMs mampu melakukan berbagai tugas umum, seperti memahami bahasa, menghasilkan teks dan gambar, dan berbicara dalam bahasa alami. Untuk informasi selengkapnya, lihat Apa itu Model Foundation.

G

AI generatif

Subset model AI yang telah dilatih pada sejumlah besar data dan yang dapat menggunakan prompt teks sederhana untuk membuat konten dan artefak baru, seperti gambar, video, teks, dan audio. Untuk informasi lebih lanjut, lihat Apa itu AI Generatif.

pemblokiran geografis

Lihat pembatasan geografis.

pembatasan geografis (pemblokiran geografis)

Di Amazon CloudFront, opsi untuk mencegah pengguna di negara tertentu mengakses distribusi konten. Anda dapat menggunakan daftar izinkan atau daftar blokir untuk menentukan negara yang disetujui dan dilarang. Untuk informasi selengkapnya, lihat Membatasi distribusi geografis konten Anda dalam dokumentasi. CloudFront

Alur kerja Gitflow

Pendekatan di mana lingkungan bawah dan atas menggunakan cabang yang berbeda dalam repositori kode sumber. Alur kerja Gitflow dianggap warisan, dan alur kerja berbasis batang adalah pendekatan modern yang lebih disukai.

gambar emas

Sebuah snapshot dari sistem atau perangkat lunak yang digunakan sebagai template untuk menyebarkan instance baru dari sistem atau perangkat lunak itu. Misalnya, di bidang manufaktur, gambar emas dapat digunakan untuk menyediakan perangkat lunak pada beberapa perangkat dan membantu meningkatkan kecepatan, skalabilitas, dan produktivitas dalam operasi manufaktur perangkat.

strategi greenfield

Tidak adanya infrastruktur yang ada di lingkungan baru. Saat mengadopsi strategi greenfield untuk arsitektur sistem, Anda dapat memilih semua teknologi baru tanpa batasan kompatibilitas dengan infrastruktur yang ada, juga dikenal sebagai brownfield. Jika Anda memperluas infrastruktur yang ada, Anda dapat memadukan strategi brownfield dan greenfield.

pagar pembatas

Aturan tingkat tinggi yang membantu mengatur sumber daya, kebijakan, dan kepatuhan di seluruh unit organisasi ()OUs. Pagar pembatas preventif menegakkan kebijakan untuk memastikan keselarasan dengan standar kepatuhan. Mereka diimplementasikan dengan menggunakan kebijakan kontrol layanan dan batas izin IAM. Detective guardrails mendeteksi pelanggaran kebijakan dan masalah kepatuhan, dan menghasilkan peringatan untuk remediasi. Mereka diimplementasikan dengan menggunakan AWS Config, AWS Security Hub, Amazon GuardDuty AWS Trusted Advisor, Amazon Inspector, dan pemeriksaan khusus AWS Lambda .

H

HA

Lihat ketersediaan tinggi.

migrasi database heterogen

Memigrasi database sumber Anda ke database target yang menggunakan mesin database yang berbeda (misalnya, Oracle ke Amazon Aurora). Migrasi heterogen biasanya merupakan bagian dari upaya arsitektur ulang, dan mengubah skema dapat menjadi tugas yang kompleks. AWS menyediakan AWS SCT yang membantu dengan konversi skema.

ketersediaan tinggi (HA)

Kemampuan beban kerja untuk beroperasi terus menerus, tanpa intervensi, jika terjadi tantangan atau bencana. Sistem HA dirancang untuk gagal secara otomatis, secara konsisten memberikan kinerja berkualitas tinggi, dan menangani beban dan kegagalan yang berbeda dengan dampak kinerja minimal.

modernisasi sejarawan

Pendekatan yang digunakan untuk memodernisasi dan meningkatkan sistem teknologi operasional (OT) untuk melayani kebutuhan industri manufaktur dengan lebih baik. Sejarawan

adalah jenis database yang digunakan untuk mengumpulkan dan menyimpan data dari berbagai sumber di pabrik.

data penahanan

Sebagian dari data historis berlabel yang ditahan dari kumpulan data yang digunakan untuk melatih model pembelajaran mesin. Anda dapat menggunakan data penahanan untuk mengevaluasi kinerja model dengan membandingkan prediksi model dengan data penahanan.

migrasi database homogen

Memigrasi database sumber Anda ke database target yang berbagi mesin database yang sama (misalnya, Microsoft SQL Server ke Amazon RDS for SQL Server). Migrasi homogen biasanya merupakan bagian dari upaya rehosting atau replatforming. Anda dapat menggunakan utilitas database asli untuk memigrasi skema.

data panas

Data yang sering diakses, seperti data real-time atau data translasi terbaru. Data ini biasanya memerlukan tingkat atau kelas penyimpanan berkinerja tinggi untuk memberikan respons kueri yang cepat.

perbaikan terbaru

Perbaikan mendesak untuk masalah kritis dalam lingkungan produksi. Karena urgensinya, perbaikan terbaru biasanya dibuat di luar alur kerja DevOps rilis biasa.

periode hypercare

Segera setelah cutover, periode waktu ketika tim migrasi mengelola dan memantau aplikasi yang dimigrasi di cloud untuk mengatasi masalah apa pun. Biasanya, periode ini panjangnya 1-4 hari. Pada akhir periode hypercare, tim migrasi biasanya mentransfer tanggung jawab untuk aplikasi ke tim operasi cloud.

I

IAc

Lihat infrastruktur sebagai kode.

kebijakan berbasis identitas

Kebijakan yang dilampirkan pada satu atau beberapa prinsip IAM yang mendefinisikan izin mereka dalam lingkungan. AWS Cloud

aplikasi idle

Aplikasi yang memiliki penggunaan CPU dan memori rata-rata antara 5 dan 20 persen selama periode 90 hari. Dalam proyek migrasi, adalah umum untuk menghentikan aplikasi ini atau mempertahankannya di tempat.

IIoT

Lihat Internet of Things industri.

infrastruktur yang tidak dapat diubah

Model yang menyebarkan infrastruktur baru untuk beban kerja produksi alih-alih memperbarui, menambal, atau memodifikasi infrastruktur yang ada. Infrastruktur yang tidak dapat diubah secara inheren lebih konsisten, andal, dan dapat diprediksi daripada infrastruktur yang dapat berubah. Untuk informasi selengkapnya, lihat praktik terbaik Deploy using immutable infrastructure di AWS Well-Architected Framework.

masuk (masuknya) VPC

Dalam arsitektur AWS multi-akun, VPC yang menerima, memeriksa, dan merutekan koneksi jaringan dari luar aplikasi. Arsitektur Referensi AWS Keamanan merekomendasikan pengaturan akun Jaringan Anda dengan inbound, outbound, dan inspeksi VPCs untuk melindungi antarmuka dua arah antara aplikasi Anda dan internet yang lebih luas.

migrasi inkremental

Strategi cutover di mana Anda memigrasikan aplikasi Anda dalam bagian-bagian kecil alih-alih melakukan satu cutover penuh. Misalnya, Anda mungkin hanya memindahkan beberapa layanan mikro atau pengguna ke sistem baru pada awalnya. Setelah Anda memverifikasi bahwa semuanya berfungsi dengan baik, Anda dapat secara bertahap memindahkan layanan mikro atau pengguna tambahan hingga Anda dapat menonaktifkan sistem lama Anda. Strategi ini mengurangi risiko yang terkait dengan migrasi besar.

Industri 4.0

Sebuah istilah yang diperkenalkan oleh Klaus Schwab pada tahun 2016 untuk merujuk pada modernisasi proses manufaktur melalui kemajuan dalam konektivitas, data real-time, otomatisasi, analitik, dan AI/ML.

infrastruktur

Semua sumber daya dan aset yang terkandung dalam lingkungan aplikasi.

infrastruktur sebagai kode (IAc)

Proses penyediaan dan pengelolaan infrastruktur aplikasi melalui satu set file konfigurasi. IAc dirancang untuk membantu Anda memusatkan manajemen infrastruktur, menstandarisasi sumber daya, dan menskalakan dengan cepat sehingga lingkungan baru dapat diulang, andal, dan konsisten.

Internet of Things industri (IIoT)

Penggunaan sensor dan perangkat yang terhubung ke internet di sektor industri, seperti manufaktur, energi, otomotif, perawatan kesehatan, ilmu kehidupan, dan pertanian. Untuk informasi lebih lanjut, lihat Membangun strategi transformasi digital Internet of Things (IIoT) industri.

inspeksi VPC

Dalam arsitektur AWS multi-akun, VPC terpusat yang mengelola inspeksi lalu lintas jaringan antara VPCs (dalam yang sama atau berbeda Wilayah AWS), internet, dan jaringan lokal. Arsitektur Referensi AWS Keamanan merekomendasikan pengaturan akun Jaringan Anda dengan inbound, outbound, dan inspeksi VPCs untuk melindungi antarmuka dua arah antara aplikasi Anda dan internet yang lebih luas.

Internet of Things (IoT)

Jaringan objek fisik yang terhubung dengan sensor atau prosesor tertanam yang berkomunikasi dengan perangkat dan sistem lain melalui internet atau melalui jaringan komunikasi lokal. Untuk informasi selengkapnya, lihat Apa itu IoT?

interpretabilitas

Karakteristik model pembelajaran mesin yang menggambarkan sejauh mana manusia dapat memahami bagaimana prediksi model bergantung pada inputnya. Untuk informasi lebih lanjut, lihat Interpretabilitas model pembelajaran mesin dengan. AWS

IoT

Lihat Internet of Things.

Perpustakaan informasi TI (ITIL)

Serangkaian praktik terbaik untuk memberikan layanan TI dan menyelaraskan layanan ini dengan persyaratan bisnis. ITIL menyediakan dasar untuk ITSM.

Manajemen layanan TI (ITSM)

Kegiatan yang terkait dengan merancang, menerapkan, mengelola, dan mendukung layanan TI untuk suatu organisasi. Untuk informasi tentang mengintegrasikan operasi cloud dengan alat ITSM, lihat panduan integrasi operasi.

ITIL

Lihat perpustakaan informasi TI.

ITSM

Lihat manajemen layanan TI.

L

kontrol akses berbasis label (LBAC)

Implementasi kontrol akses wajib (MAC) di mana pengguna dan data itu sendiri masing-masing secara eksplisit diberi nilai label keamanan. Persimpangan antara label keamanan pengguna dan label keamanan data menentukan baris dan kolom mana yang dapat dilihat oleh pengguna.

landing zone

Landing zone adalah AWS lingkungan multi-akun yang dirancang dengan baik yang dapat diskalakan dan aman. Ini adalah titik awal dari mana organisasi Anda dapat dengan cepat meluncurkan dan menyebarkan beban kerja dan aplikasi dengan percaya diri dalam lingkungan keamanan dan infrastruktur mereka. Untuk informasi selengkapnya tentang zona pendaratan, lihat Menyiapkan lingkungan multi-akun AWS yang aman dan dapat diskalakan.

model bahasa besar (LLM)

Model AI pembelajaran mendalam yang dilatih sebelumnya pada sejumlah besar data. LLM dapat melakukan beberapa tugas, seperti menjawab pertanyaan, meringkas dokumen, menerjemahkan teks ke dalam bahasa lain, dan menyelesaikan kalimat. Untuk informasi lebih lanjut, lihat Apa itu LLMs.

migrasi besar

Migrasi 300 atau lebih server.

LBAC

Lihat kontrol akses berbasis label.

hak istimewa paling sedikit

Praktik keamanan terbaik untuk memberikan izin minimum yang diperlukan untuk melakukan tugas. Untuk informasi selengkapnya, lihat Menerapkan izin hak istimewa terkecil dalam dokumentasi IAM.

angkat dan geser

Lihat 7 Rs.

sistem endian kecil

Sebuah sistem yang menyimpan byte paling tidak signifikan terlebih dahulu. Lihat juga endianness.

LLM

Lihat model bahasa besar.

lingkungan yang lebih rendah

Lihat lingkungan.

M

pembelajaran mesin (ML)

Jenis kecerdasan buatan yang menggunakan algoritma dan teknik untuk pengenalan pola dan pembelajaran. ML menganalisis dan belajar dari data yang direkam, seperti data Internet of Things (IoT), untuk menghasilkan model statistik berdasarkan pola. Untuk informasi selengkapnya, lihat Machine Learning.

cabang utama

Lihat cabang.

malware

Perangkat lunak yang dirancang untuk membahayakan keamanan atau privasi komputer. Malware dapat mengganggu sistem komputer, membocorkan informasi sensitif, atau mendapatkan akses yang tidak sah. Contoh malware termasuk virus, worm, ransomware, Trojan horse, spyware, dan keyloggers.

layanan terkelola

Layanan AWS yang AWS mengoperasikan lapisan infrastruktur, sistem operasi, dan platform, dan Anda mengakses titik akhir untuk menyimpan dan mengambil data. Amazon Simple Storage Service (Amazon S3) dan Amazon DynamoDB adalah contoh layanan terkelola. Ini juga dikenal sebagai layanan abstrak.

sistem eksekusi manufaktur (MES)

Sistem perangkat lunak untuk melacak, memantau, mendokumentasikan, dan mengendalikan proses produksi yang mengubah bahan baku menjadi produk jadi di lantai toko.

PETA

Lihat Program Percepatan Migrasi.

mekanisme

Proses lengkap di mana Anda membuat alat, mendorong adopsi alat, dan kemudian memeriksa hasilnya untuk melakukan penyesuaian. Mekanisme adalah siklus yang memperkuat dan meningkatkan dirinya sendiri saat beroperasi. Untuk informasi lebih lanjut, lihat Membangun mekanisme di AWS Well-Architected Framework.

akun anggota

Semua Akun AWS selain akun manajemen yang merupakan bagian dari organisasi di AWS Organizations. Akun dapat menjadi anggota dari hanya satu organisasi pada suatu waktu.

MES

Lihat sistem eksekusi manufaktur.

Transportasi Telemetri Antrian Pesan (MQTT)

Protokol komunikasi ringan machine-to-machine (M2M), berdasarkan pola terbitkan/berlangganan, untuk perangkat IoT yang dibatasi sumber daya.

layanan mikro

Layanan kecil dan independen yang berkomunikasi dengan jelas APIs dan biasanya dimiliki oleh tim kecil yang mandiri. Misalnya, sistem asuransi mungkin mencakup layanan mikro yang memetakan kemampuan bisnis, seperti penjualan atau pemasaran, atau subdomain, seperti pembelian, klaim, atau analitik. Manfaat layanan mikro termasuk kelincahan, penskalaan yang fleksibel, penyebaran yang mudah, kode yang dapat digunakan kembali, dan ketahanan. Untuk

informasi selengkapnya, lihat Mengintegrasikan layanan mikro dengan menggunakan layanan tanpa AWS server.

arsitektur microservices

Pendekatan untuk membangun aplikasi dengan komponen independen yang menjalankan setiap proses aplikasi sebagai layanan mikro. Layanan mikro ini berkomunikasi melalui antarmuka yang terdefinisi dengan baik dengan menggunakan ringan. APIs Setiap layanan mikro dalam arsitektur ini dapat diperbarui, digunakan, dan diskalakan untuk memenuhi permintaan fungsi tertentu dari suatu aplikasi. Untuk informasi selengkapnya, lihat Menerapkan layanan mikro di AWS.

Program Percepatan Migrasi (MAP)

AWS Program yang menyediakan dukungan konsultasi, pelatihan, dan layanan untuk membantu organisasi membangun fondasi operasional yang kuat untuk pindah ke cloud, dan untuk membantu mengimbangi biaya awal migrasi. MAP mencakup metodologi migrasi untuk mengeksekusi migrasi lama dengan cara metodis dan seperangkat alat untuk mengotomatisasi dan mempercepat skenario migrasi umum.

migrasi dalam skala

Proses memindahkan sebagian besar portofolio aplikasi ke cloud dalam gelombang, dengan lebih banyak aplikasi bergerak pada tingkat yang lebih cepat di setiap gelombang. Fase ini menggunakan praktik dan pelajaran terbaik dari fase sebelumnya untuk mengimplementasikan pabrik migrasi tim, alat, dan proses untuk merampingkan migrasi beban kerja melalui otomatisasi dan pengiriman tangkas. Ini adalah fase ketiga dari strategi AWS migrasi.

pabrik migrasi

Tim lintas fungsi yang merampingkan migrasi beban kerja melalui pendekatan otomatis dan gesit. Tim pabrik migrasi biasanya mencakup operasi, analis dan pemilik bisnis, insinyur migrasi, pengembang, dan DevOps profesional yang bekerja di sprint. Antara 20 dan 50 persen portofolio aplikasi perusahaan terdiri dari pola berulang yang dapat dioptimalkan dengan pendekatan pabrik. Untuk informasi selengkapnya, lihat diskusi tentang pabrik migrasi dan panduan Pabrik Migrasi Cloud di kumpulan konten ini.

metadata migrasi

Informasi tentang aplikasi dan server yang diperlukan untuk menyelesaikan migrasi. Setiap pola migrasi memerlukan satu set metadata migrasi yang berbeda. Contoh metadata migrasi termasuk subnet target, grup keamanan, dan akun. AWS

pola migrasi

Tugas migrasi berulang yang merinci strategi migrasi, tujuan migrasi, dan aplikasi atau layanan migrasi yang digunakan. Contoh: Rehost migrasi ke Amazon EC2 dengan Layanan Migrasi AWS Aplikasi.

Penilaian Portofolio Migrasi (MPA)

Alat online yang menyediakan informasi untuk memvalidasi kasus bisnis untuk bermigrasi ke. AWS Cloud MPA menyediakan penilaian portofolio terperinci (ukuran kanan server, harga, perbandingan TCO, analisis biaya migrasi) serta perencanaan migrasi (analisis data aplikasi dan pengumpulan data, pengelompokan aplikasi, prioritas migrasi, dan perencanaan gelombang). Alat MPA (memerlukan login) tersedia gratis untuk semua AWS konsultan dan konsultan APN Partner.

Penilaian Kesiapan Migrasi (MRA)

Proses mendapatkan wawasan tentang status kesiapan cloud organisasi, mengidentifikasi kekuatan dan kelemahan, dan membangun rencana aksi untuk menutup kesenjangan yang diidentifikasi, menggunakan CAF. AWS Untuk informasi selengkapnya, lihat panduan kesiapan migrasi. MRA adalah tahap pertama dari strategi AWS migrasi.

strategi migrasi

Pendekatan yang digunakan untuk memigrasikan beban kerja ke file. AWS Cloud Untuk informasi lebih lanjut, lihat entri 7 Rs di glosarium ini dan lihat Memobilisasi organisasi Anda untuk mempercepat migrasi skala besar.

ML

Lihat pembelajaran mesin.

modernisasi

Mengubah aplikasi usang (warisan atau monolitik) dan infrastrukturnya menjadi sistem yang gesit, elastis, dan sangat tersedia di cloud untuk mengurangi biaya, mendapatkan efisiensi, dan memanfaatkan inovasi. Untuk informasi selengkapnya, lihat Strategi untuk memodernisasi aplikasi di. AWS Cloud

penilaian kesiapan modernisasi

Evaluasi yang membantu menentukan kesiapan modernisasi aplikasi organisasi; mengidentifikasi manfaat, risiko, dan dependensi; dan menentukan seberapa baik organisasi dapat mendukung keadaan masa depan aplikasi tersebut. Hasil penilaian adalah cetak biru arsitektur target, peta

jalan yang merinci fase pengembangan dan tonggak untuk proses modernisasi, dan rencana aksi untuk mengatasi kesenjangan yang diidentifikasi. Untuk informasi lebih lanjut, lihat Mengevaluasi kesiapan modernisasi untuk aplikasi di. AWS Cloud

aplikasi monolitik (monolit)

Aplikasi yang berjalan sebagai layanan tunggal dengan proses yang digabungkan secara ketat. Aplikasi monolitik memiliki beberapa kelemahan. Jika satu fitur aplikasi mengalami lonjakan permintaan, seluruh arsitektur harus diskalakan. Menambahkan atau meningkatkan fitur aplikasi monolitik juga menjadi lebih kompleks ketika basis kode tumbuh. Untuk mengatasi masalah ini, Anda dapat menggunakan arsitektur microservices. Untuk informasi lebih lanjut, lihat Menguraikan monolit menjadi layanan mikro.

MPA

Lihat Penilaian Portofolio Migrasi.

MQTT

Lihat Transportasi Telemetri Antrian Pesan.

klasifikasi multiclass

Sebuah proses yang membantu menghasilkan prediksi untuk beberapa kelas (memprediksi satu dari lebih dari dua hasil). Misalnya, model ML mungkin bertanya “Apakah produk ini buku, mobil, atau telepon?” atau “Kategori produk mana yang paling menarik bagi pelanggan ini?”

infrastruktur yang bisa berubah

Model yang memperbarui dan memodifikasi infrastruktur yang ada untuk beban kerja produksi. Untuk meningkatkan konsistensi, keandalan, dan prediktabilitas, AWS Well-Architected Framework merekomendasikan penggunaan infrastruktur yang tidak dapat diubah sebagai praktik terbaik.

O

OAC

Lihat kontrol akses asal.

OAI

Lihat identitas akses asal.

OCM

Lihat manajemen perubahan organisasi.

migrasi offline

Metode migrasi di mana beban kerja sumber diturunkan selama proses migrasi. Metode ini melibatkan waktu henti yang diperpanjang dan biasanya digunakan untuk beban kerja kecil dan tidak kritis.

OI

Lihat integrasi operasi.

OLA

Lihat perjanjian tingkat operasional.

migrasi online

Metode migrasi di mana beban kerja sumber disalin ke sistem target tanpa diambil offline. Aplikasi yang terhubung ke beban kerja dapat terus berfungsi selama migrasi. Metode ini melibatkan waktu henti nol hingga minimal dan biasanya digunakan untuk beban kerja produksi yang kritis.

OPC-UA

Lihat Komunikasi Proses Terbuka - Arsitektur Terpadu.

Komunikasi Proses Terbuka - Arsitektur Terpadu (OPC-UA)

Protokol komunikasi machine-to-machine (M2M) untuk otomasi industri. OPC-UA menyediakan standar interoperabilitas dengan enkripsi data, otentikasi, dan skema otorisasi.

perjanjian tingkat operasional (OLA)

Perjanjian yang menjelaskan apa yang dijanjikan kelompok TI fungsional untuk diberikan satu sama lain, untuk mendukung perjanjian tingkat layanan (SLA).

Tinjauan Kesiapan Operasional (ORR)

Daftar pertanyaan dan praktik terbaik terkait yang membantu Anda memahami, mengevaluasi, mencegah, atau mengurangi ruang lingkup insiden dan kemungkinan kegagalan. Untuk informasi lebih lanjut, lihat Ulasan Kesiapan Operasional (ORR) dalam Kerangka Kerja Well-Architected AWS .

teknologi operasional (OT)

Sistem perangkat keras dan perangkat lunak yang bekerja dengan lingkungan fisik untuk mengendalikan operasi industri, peralatan, dan infrastruktur. Di bidang manufaktur, integrasi sistem OT dan teknologi informasi (TI) adalah fokus utama untuk transformasi Industri 4.0.

integrasi operasi (OI)

Proses modernisasi operasi di cloud, yang melibatkan perencanaan kesiapan, otomatisasi, dan integrasi. Untuk informasi selengkapnya, lihat panduan integrasi operasi.

jejak organisasi

Jejak yang dibuat oleh AWS CloudTrail itu mencatat semua peristiwa untuk semua Akun AWS dalam organisasi di AWS Organizations. Jejak ini dibuat di setiap Akun AWS bagian organisasi dan melacak aktivitas di setiap akun. Untuk informasi selengkapnya, lihat Membuat jejak untuk organisasi dalam CloudTrail dokumentasi.

manajemen perubahan organisasi (OCM)

Kerangka kerja untuk mengelola transformasi bisnis utama yang mengganggu dari perspektif orang, budaya, dan kepemimpinan. OCM membantu organisasi mempersiapkan, dan transisi ke, sistem dan strategi baru dengan mempercepat adopsi perubahan, mengatasi masalah transisi, dan mendorong perubahan budaya dan organisasi. Dalam strategi AWS migrasi, kerangka kerja ini disebut percepatan orang, karena kecepatan perubahan yang diperlukan dalam proyek adopsi cloud. Untuk informasi lebih lanjut, lihat panduan OCM.

kontrol akses asal (OAC)

Di CloudFront, opsi yang disempurnakan untuk membatasi akses untuk mengamankan konten Amazon Simple Storage Service (Amazon S3) Anda. OAC mendukung semua bucket S3 di semua Wilayah AWS, enkripsi sisi server dengan AWS KMS (SSE-KMS), dan dinamis dan permintaan ke bucket S3. `PUT` `DELETE`

identitas akses asal (OAI)

Di CloudFront, opsi untuk membatasi akses untuk mengamankan konten Amazon S3 Anda. Saat Anda menggunakan OAI, CloudFront buat prinsipal yang dapat diautentikasi oleh Amazon S3. Prinsipal yang diautentikasi dapat mengakses konten dalam bucket S3 hanya melalui distribusi tertentu. CloudFront Lihat juga OAC, yang menyediakan kontrol akses yang lebih terperinci dan ditingkatkan.

ORR

Lihat tinjauan kesiapan operasional.

OT

Lihat teknologi operasional.

keluar (jalan keluar) VPC

Dalam arsitektur AWS multi-akun, VPC yang menangani koneksi jaringan yang dimulai dari dalam aplikasi. Arsitektur Referensi AWS Keamanan merekomendasikan pengaturan akun Jaringan Anda dengan inbound, outbound, dan inspeksi VPCs untuk melindungi antarmuka dua arah antara aplikasi Anda dan internet yang lebih luas.

P

batas izin

Kebijakan manajemen IAM yang dilampirkan pada prinsipal IAM untuk menetapkan izin maksimum yang dapat dimiliki pengguna atau peran. Untuk informasi selengkapnya, lihat Batas izin dalam dokumentasi IAM.

Informasi Identifikasi Pribadi (PII)

Informasi yang, jika dilihat secara langsung atau dipasangkan dengan data terkait lainnya, dapat digunakan untuk menyimpulkan identitas individu secara wajar. Contoh PII termasuk nama, alamat, dan informasi kontak.

PII

Lihat informasi yang dapat diidentifikasi secara pribadi.

buku pedoman

Serangkaian langkah yang telah ditentukan sebelumnya yang menangkap pekerjaan yang terkait dengan migrasi, seperti mengirimkan fungsi operasi inti di cloud. Buku pedoman dapat berupa skrip, runbook otomatis, atau ringkasan proses atau langkah-langkah yang diperlukan untuk mengoperasikan lingkungan modern Anda.

PLC

Lihat pengontrol logika yang dapat diprogram.

PLM

Lihat manajemen siklus hidup produk.

kebijakan

Objek yang dapat menentukan izin (lihat kebijakan berbasis identitas), menentukan kondisi akses (lihat kebijakan berbasis sumber daya), atau menentukan izin maksimum untuk semua akun di organisasi (lihat kebijakan kontrol layanan). AWS Organizations

ketekunan poliglot

Secara independen memilih teknologi penyimpanan data microservice berdasarkan pola akses data dan persyaratan lainnya. Jika layanan mikro Anda memiliki teknologi penyimpanan data yang sama, mereka dapat menghadapi tantangan implementasi atau mengalami kinerja yang buruk. Layanan mikro lebih mudah diimplementasikan dan mencapai kinerja dan skalabilitas yang lebih baik jika mereka menggunakan penyimpanan data yang paling sesuai dengan kebutuhan mereka. Untuk informasi selengkapnya, lihat Mengaktifkan persistensi data di layanan mikro.

penilaian portofolio

Proses menemukan, menganalisis, dan memprioritaskan portofolio aplikasi untuk merencanakan migrasi. Untuk informasi selengkapnya, lihat Mengevaluasi kesiapan migrasi.

predikat

Kondisi kueri yang mengembalikan `true` atau`false`, biasanya terletak di `WHERE` klausa.

predikat pushdown

Teknik optimasi kueri database yang menyaring data dalam kueri sebelum transfer. Ini mengurangi jumlah data yang harus diambil dan diproses dari database relasional, dan meningkatkan kinerja kueri.

kontrol preventif

Kontrol keamanan yang dirancang untuk mencegah suatu peristiwa terjadi. Kontrol ini adalah garis pertahanan pertama untuk membantu mencegah akses tidak sah atau perubahan yang tidak diinginkan ke jaringan Anda. Untuk informasi selengkapnya, lihat Kontrol pencegahan dalam Menerapkan kontrol keamanan pada. AWS

principal

Entitas AWS yang dapat melakukan tindakan dan mengakses sumber daya. Entitas ini biasanya merupakan pengguna root untuk Akun AWS, peran IAM, atau pengguna. Untuk informasi selengkapnya, lihat Prinsip dalam istilah dan konsep Peran dalam dokumentasi IAM.

privasi berdasarkan desain

Pendekatan rekayasa sistem yang memperhitungkan privasi melalui seluruh proses pengembangan.

zona yang dihosting pribadi

Container yang menyimpan informasi tentang bagaimana Anda ingin Amazon Route 53 merespons kueri DNS untuk domain dan subdomainnya dalam satu atau lebih. VPCs Untuk informasi selengkapnya, lihat Bekerja dengan zona yang dihosting pribadi di dokumentasi Route 53.

kontrol proaktif

Kontrol keamanan yang dirancang untuk mencegah penyebaran sumber daya yang tidak sesuai. Kontrol ini memindai sumber daya sebelum disediakan. Jika sumber daya tidak sesuai dengan kontrol, maka itu tidak disediakan. Untuk informasi selengkapnya, lihat panduan referensi Kontrol dalam AWS Control Tower dokumentasi dan lihat Kontrol proaktif dalam Menerapkan kontrol keamanan pada AWS.

manajemen siklus hidup produk (PLM)

Manajemen data dan proses untuk suatu produk di seluruh siklus hidupnya, mulai dari desain, pengembangan, dan peluncuran, melalui pertumbuhan dan kematangan, hingga penurunan dan penghapusan.

lingkungan produksi

Lihat lingkungan.

pengontrol logika yang dapat diprogram (PLC)

Di bidang manufaktur, komputer yang sangat andal dan mudah beradaptasi yang memantau mesin dan mengotomatiskan proses manufaktur.

rantai cepat

Menggunakan output dari satu prompt LLM sebagai input untuk prompt berikutnya untuk menghasilkan respons yang lebih baik. Teknik ini digunakan untuk memecah tugas yang kompleks menjadi subtugas, atau untuk secara iteratif memperbaiki atau memperluas respons awal. Ini membantu meningkatkan akurasi dan relevansi respons model dan memungkinkan hasil yang lebih terperinci dan dipersonalisasi.

pseudonimisasi

Proses penggantian pengenal pribadi dalam kumpulan data dengan nilai placeholder. Pseudonimisasi dapat membantu melindungi privasi pribadi. Data pseudonim masih dianggap sebagai data pribadi.

publish/subscribe (pub/sub)

Pola yang memungkinkan komunikasi asinkron antara layanan mikro untuk meningkatkan skalabilitas dan daya tanggap. Misalnya, dalam MES berbasis layanan mikro, layanan mikro dapat mempublikasikan pesan peristiwa ke saluran yang dapat berlangganan layanan mikro lainnya. Sistem dapat menambahkan layanan mikro baru tanpa mengubah layanan penerbitan.

Q

rencana kueri

Serangkaian langkah, seperti instruksi, yang digunakan untuk mengakses data dalam sistem database relasional SQL.

regresi rencana kueri

Ketika pengoptimal layanan database memilih rencana yang kurang optimal daripada sebelum perubahan yang diberikan ke lingkungan database. Hal ini dapat disebabkan oleh perubahan statistik, kendala, pengaturan lingkungan, pengikatan parameter kueri, dan pembaruan ke mesin database.

R

Matriks RACI

Lihat bertanggung jawab, akuntabel, dikonsultasikan, diinformasikan (RACI).

LAP

Lihat Retrieval Augmented Generation.

ransomware

Perangkat lunak berbahaya yang dirancang untuk memblokir akses ke sistem komputer atau data sampai pembayaran dilakukan.

Matriks RASCI

Lihat bertanggung jawab, akuntabel, dikonsultasikan, diinformasikan (RACI).

RCAC

Lihat kontrol akses baris dan kolom.

replika baca

Salinan database yang digunakan untuk tujuan read-only. Anda dapat merutekan kueri ke replika baca untuk mengurangi beban pada database utama Anda.

arsitek ulang

Lihat 7 Rs.

tujuan titik pemulihan (RPO)

Jumlah waktu maksimum yang dapat diterima sejak titik pemulihan data terakhir. Ini menentukan apa yang dianggap sebagai kehilangan data yang dapat diterima antara titik pemulihan terakhir dan gangguan layanan.

tujuan waktu pemulihan (RTO)

Penundaan maksimum yang dapat diterima antara gangguan layanan dan pemulihan layanan.

refactor

Lihat 7 Rs.

Wilayah

Kumpulan AWS sumber daya di wilayah geografis. Masing-masing Wilayah AWS terisolasi dan independen dari yang lain untuk memberikan toleransi kesalahan, stabilitas, dan ketahanan. Untuk informasi selengkapnya, lihat Menentukan Wilayah AWS akun yang dapat digunakan.

regresi

Teknik ML yang memprediksi nilai numerik. Misalnya, untuk memecahkan masalah “Berapa harga rumah ini akan dijual?” Model ML dapat menggunakan model regresi linier untuk memprediksi harga jual rumah berdasarkan fakta yang diketahui tentang rumah (misalnya, luas persegi).

rehost

Lihat 7 Rs.

melepaskan

Dalam proses penyebaran, tindakan mempromosikan perubahan pada lingkungan produksi.

memindahkan

Lihat 7 Rs.

memplatform ulang

Lihat 7 Rs.

pembelian kembali

Lihat 7 Rs.

ketahanan

Kemampuan aplikasi untuk melawan atau pulih dari gangguan. Ketersediaan tinggi dan pemulihan bencana adalah pertimbangan umum ketika merencanakan ketahanan di. AWS Cloud Untuk informasi lebih lanjut, lihat AWS Cloud Ketahanan.

kebijakan berbasis sumber daya

Kebijakan yang dilampirkan ke sumber daya, seperti bucket Amazon S3, titik akhir, atau kunci enkripsi. Jenis kebijakan ini menentukan prinsipal mana yang diizinkan mengakses, tindakan yang didukung, dan kondisi lain yang harus dipenuhi.

matriks yang bertanggung jawab, akuntabel, dikonsultasikan, diinformasikan (RACI)

Matriks yang mendefinisikan peran dan tanggung jawab untuk semua pihak yang terlibat dalam kegiatan migrasi dan operasi cloud. Nama matriks berasal dari jenis tanggung jawab yang didefinisikan dalam matriks: bertanggung jawab (R), akuntabel (A), dikonsultasikan (C), dan diinformasikan (I). Tipe dukungan (S) adalah opsional. Jika Anda menyertakan dukungan, matriks disebut matriks RASCI, dan jika Anda mengecualikannya, itu disebut matriks RACI.

kontrol responsif

Kontrol keamanan yang dirancang untuk mendorong remediasi efek samping atau penyimpangan dari garis dasar keamanan Anda. Untuk informasi selengkapnya, lihat Kontrol responsif dalam Menerapkan kontrol keamanan pada AWS.

melestarikan

Lihat 7 Rs.

pensiun

Lihat 7 Rs.

Retrieval Augmented Generation (RAG)

Teknologi AI generatif di mana LLM merujuk sumber data otoritatif yang berada di luar sumber data pelatihannya sebelum menghasilkan respons. Misalnya, model RAG mungkin melakukan pencarian semantik dari basis pengetahuan organisasi atau data kustom. Untuk informasi lebih lanjut, lihat Apa itu RAG.

rotasi

Proses memperbarui rahasia secara berkala untuk membuatnya lebih sulit bagi penyerang untuk mengakses kredensil.

kontrol akses baris dan kolom (RCAC)

Penggunaan ekspresi SQL dasar dan fleksibel yang telah menetapkan aturan akses. RCAC terdiri dari izin baris dan topeng kolom.

RPO

Lihat tujuan titik pemulihan.

RTO

Lihat tujuan waktu pemulihan.

buku runbook

Satu set prosedur manual atau otomatis yang diperlukan untuk melakukan tugas tertentu. Ini biasanya dibangun untuk merampingkan operasi berulang atau prosedur dengan tingkat kesalahan yang tinggi.

D

SAML 2.0

Standar terbuka yang digunakan oleh banyak penyedia identitas (IdPs). Fitur ini memungkinkan sistem masuk tunggal gabungan (SSO), sehingga pengguna dapat masuk ke AWS Management Console atau memanggil operasi AWS API tanpa Anda harus membuat pengguna di IAM untuk semua orang di organisasi Anda. Untuk informasi lebih lanjut tentang federasi berbasis SAMP 2.0, lihat Tentang federasi berbasis SAMP 2.0 dalam dokumentasi IAM.

SCADA

Lihat kontrol pengawasan dan akuisisi data.

SCP

Lihat kebijakan kontrol layanan.

Rahasia

Dalam AWS Secrets Manager, informasi rahasia atau terbatas, seperti kata sandi atau kredensi pengguna, yang Anda simpan dalam bentuk terenkripsi. Ini terdiri dari nilai rahasia dan metadata-nya. Nilai rahasia dapat berupa biner, string tunggal, atau beberapa string. Untuk informasi selengkapnya, lihat Apa yang ada di rahasia Secrets Manager? dalam dokumentasi Secrets Manager.

keamanan dengan desain

Pendekatan rekayasa sistem yang memperhitungkan keamanan melalui seluruh proses pengembangan.

kontrol keamanan

Pagar pembatas teknis atau administratif yang mencegah, mendeteksi, atau mengurangi kemampuan pelaku ancaman untuk mengeksploitasi kerentanan keamanan. Ada empat jenis kontrol keamanan utama: preventif, detektif, responsif, dan proaktif.

pengerasan keamanan

Proses mengurangi permukaan serangan untuk membuatnya lebih tahan terhadap serangan. Ini dapat mencakup tindakan seperti menghapus sumber daya yang tidak lagi diperlukan, menerapkan praktik keamanan terbaik untuk memberikan hak istimewa paling sedikit, atau menonaktifkan fitur yang tidak perlu dalam file konfigurasi.

sistem informasi keamanan dan manajemen acara (SIEM)

Alat dan layanan yang menggabungkan sistem manajemen informasi keamanan (SIM) dan manajemen acara keamanan (SEM). Sistem SIEM mengumpulkan, memantau, dan menganalisis data dari server, jaringan, perangkat, dan sumber lain untuk mendeteksi ancaman dan pelanggaran keamanan, dan untuk menghasilkan peringatan.

otomatisasi respons keamanan

Tindakan yang telah ditentukan dan diprogram yang dirancang untuk secara otomatis merespons atau memulihkan peristiwa keamanan. Otomatisasi ini berfungsi sebagai kontrol keamanan

detektif atau responsif yang membantu Anda menerapkan praktik terbaik AWS keamanan. Contoh tindakan respons otomatis termasuk memodifikasi grup keamanan VPC, menambal instans EC2 Amazon, atau memutar kredensil.

enkripsi sisi server

Enkripsi data di tujuannya, oleh Layanan AWS yang menerimanya.

kebijakan kontrol layanan (SCP)

Kebijakan yang menyediakan kontrol terpusat atas izin untuk semua akun di organisasi. AWS Organizations SCPs menentukan pagar pembatas atau menetapkan batasan pada tindakan yang dapat didelegasikan oleh administrator kepada pengguna atau peran. Anda dapat menggunakan SCPs daftar izin atau daftar penolakan, untuk menentukan layanan atau tindakan mana yang diizinkan atau dilarang. Untuk informasi selengkapnya, lihat Kebijakan kontrol layanan dalam AWS Organizations dokumentasi.

titik akhir layanan

URL titik masuk untuk file Layanan AWS. Anda dapat menggunakan endpoint untuk terhubung secara terprogram ke layanan target. Untuk informasi selengkapnya, lihat Layanan AWS titik akhir di Referensi Umum AWS.

perjanjian tingkat layanan (SLA)

Perjanjian yang menjelaskan apa yang dijanjikan tim TI untuk diberikan kepada pelanggan mereka, seperti waktu kerja dan kinerja layanan.

indikator tingkat layanan (SLI)

Pengukuran aspek kinerja layanan, seperti tingkat kesalahan, ketersediaan, atau throughputnya.

tujuan tingkat layanan (SLO)

Metrik target yang mewakili kesehatan layanan, yang diukur dengan indikator tingkat layanan.

model tanggung jawab bersama

Model yang menjelaskan tanggung jawab yang Anda bagikan AWS untuk keamanan dan kepatuhan cloud. AWS bertanggung jawab atas keamanan cloud, sedangkan Anda bertanggung jawab atas keamanan di cloud. Untuk informasi selengkapnya, lihat Model tanggung jawab bersama.

SIEM

Lihat informasi keamanan dan sistem manajemen acara.

titik kegagalan tunggal (SPOF)

Kegagalan dalam satu komponen penting dari aplikasi yang dapat mengganggu sistem.

SLA

Lihat perjanjian tingkat layanan.

SLI

Lihat indikator tingkat layanan.

SLO

Lihat tujuan tingkat layanan.

split-and-seed model

Pola untuk menskalakan dan mempercepat proyek modernisasi. Ketika fitur baru dan rilis produk didefinisikan, tim inti berpisah untuk membuat tim produk baru. Ini membantu meningkatkan kemampuan dan layanan organisasi Anda, meningkatkan produktivitas pengembang, dan mendukung inovasi yang cepat. Untuk informasi lebih lanjut, lihat Pendekatan bertahap untuk memodernisasi aplikasi di. AWS Cloud

SPOF

Lihat satu titik kegagalan.

skema bintang

Struktur organisasi database yang menggunakan satu tabel fakta besar untuk menyimpan data transaksional atau terukur dan menggunakan satu atau lebih tabel dimensi yang lebih kecil untuk menyimpan atribut data. Struktur ini dirancang untuk digunakan dalam gudang data atau untuk tujuan intelijen bisnis.

pola ara pencekik

Pendekatan untuk memodernisasi sistem monolitik dengan menulis ulang secara bertahap dan mengganti fungsionalitas sistem sampai sistem warisan dapat dinonaktifkan. Pola ini menggunakan analogi pohon ara yang tumbuh menjadi pohon yang sudah mapan dan akhirnya mengatasi dan menggantikan inangnya. Pola ini diperkenalkan oleh Martin Fowler sebagai cara untuk mengelola risiko saat menulis ulang sistem monolitik. Untuk contoh cara menerapkan pola ini, lihat Memodernisasi layanan web Microsoft ASP.NET (ASMX) lama secara bertahap menggunakan container dan Amazon API Gateway.

subnet

Rentang alamat IP dalam VPC Anda. Subnet harus berada di Availability Zone tunggal.

kontrol pengawasan dan akuisisi data (SCADA)

Di bidang manufaktur, sistem yang menggunakan perangkat keras dan perangkat lunak untuk memantau aset fisik dan operasi produksi.

enkripsi simetris

Algoritma enkripsi yang menggunakan kunci yang sama untuk mengenkripsi dan mendekripsi data.

pengujian sintetis

Menguji sistem dengan cara yang mensimulasikan interaksi pengguna untuk mendeteksi potensi masalah atau untuk memantau kinerja. Anda dapat menggunakan Amazon CloudWatch Synthetics untuk membuat tes ini.

sistem prompt

Teknik untuk memberikan konteks, instruksi, atau pedoman ke LLM untuk mengarahkan perilakunya. Permintaan sistem membantu mengatur konteks dan menetapkan aturan untuk interaksi dengan pengguna.

T

tag

Pasangan nilai kunci yang bertindak sebagai metadata untuk mengatur sumber daya Anda. AWS Tanda dapat membantu Anda mengelola, mengidentifikasi, mengatur, dan memfilter sumber daya. Untuk informasi selengkapnya, lihat Menandai AWS sumber daya Anda.

variabel target

Nilai yang Anda coba prediksi dalam ML yang diawasi. Ini juga disebut sebagai variabel hasil. Misalnya, dalam pengaturan manufaktur, variabel target bisa menjadi cacat produk.

daftar tugas

Alat yang digunakan untuk melacak kemajuan melalui runbook. Daftar tugas berisi ikhtisar runbook dan daftar tugas umum yang harus diselesaikan. Untuk setiap tugas umum, itu termasuk perkiraan jumlah waktu yang dibutuhkan, pemilik, dan kemajuan.

lingkungan uji

Lihat lingkungan.

pelatihan

Untuk menyediakan data bagi model ML Anda untuk dipelajari. Data pelatihan harus berisi jawaban yang benar. Algoritma pembelajaran menemukan pola dalam data pelatihan yang memetakan atribut data input ke target (jawaban yang ingin Anda prediksi). Ini menghasilkan model ML yang menangkap pola-pola ini. Anda kemudian dapat menggunakan model ML untuk membuat prediksi pada data baru yang Anda tidak tahu targetnya.

gerbang transit

Hub transit jaringan yang dapat Anda gunakan untuk menghubungkan jaringan Anda VPCs dan lokal. Untuk informasi selengkapnya, lihat Apa itu gateway transit dalam AWS Transit Gateway dokumentasi.

alur kerja berbasis batang

Pendekatan di mana pengembang membangun dan menguji fitur secara lokal di cabang fitur dan kemudian menggabungkan perubahan tersebut ke cabang utama. Cabang utama kemudian dibangun untuk pengembangan, praproduksi, dan lingkungan produksi, secara berurutan.

akses tepercaya

Memberikan izin ke layanan yang Anda tentukan untuk melakukan tugas di organisasi Anda di dalam AWS Organizations dan di akunnya atas nama Anda. Layanan tepercaya menciptakan peran terkait layanan di setiap akun, ketika peran itu diperlukan, untuk melakukan tugas manajemen untuk Anda. Untuk informasi selengkapnya, lihat Menggunakan AWS Organizations dengan AWS layanan lain dalam AWS Organizations dokumentasi.

penyetelan

Untuk mengubah aspek proses pelatihan Anda untuk meningkatkan akurasi model ML. Misalnya, Anda dapat melatih model ML dengan membuat set pelabelan, menambahkan label, dan kemudian mengulangi langkah-langkah ini beberapa kali di bawah pengaturan yang berbeda untuk mengoptimalkan model.

tim dua pizza

Sebuah DevOps tim kecil yang bisa Anda beri makan dengan dua pizza. Ukuran tim dua pizza memastikan peluang terbaik untuk berkolaborasi dalam pengembangan perangkat lunak.

U

waswas

Sebuah konsep yang mengacu pada informasi yang tidak tepat, tidak lengkap, atau tidak diketahui yang dapat merusak keandalan model ML prediktif. Ada dua jenis ketidakpastian: ketidakpastian epistemik disebabkan oleh data yang terbatas dan tidak lengkap, sedangkan ketidakpastian aleatorik disebabkan oleh kebisingan dan keacakan yang melekat dalam data. Untuk informasi lebih lanjut, lihat panduan Mengukur ketidakpastian dalam sistem pembelajaran mendalam.

tugas yang tidak terdiferensiasi

Juga dikenal sebagai angkat berat, pekerjaan yang diperlukan untuk membuat dan mengoperasikan aplikasi tetapi itu tidak memberikan nilai langsung kepada pengguna akhir atau memberikan keunggulan kompetitif. Contoh tugas yang tidak terdiferensiasi termasuk pengadaan, pemeliharaan, dan perencanaan kapasitas.

lingkungan atas

Lihat lingkungan.

V

menyedot debu

Operasi pemeliharaan database yang melibatkan pembersihan setelah pembaruan tambahan untuk merebut kembali penyimpanan dan meningkatkan kinerja.

kendali versi

Proses dan alat yang melacak perubahan, seperti perubahan kode sumber dalam repositori.

Peering VPC

Koneksi antara dua VPCs yang memungkinkan Anda untuk merutekan lalu lintas dengan menggunakan alamat IP pribadi. Untuk informasi selengkapnya, lihat Apa itu peering VPC di dokumentasi VPC Amazon.

kerentanan

Kelemahan perangkat lunak atau perangkat keras yang membahayakan keamanan sistem.

W

cache hangat

Cache buffer yang berisi data saat ini dan relevan yang sering diakses. Instance database dapat membaca dari cache buffer, yang lebih cepat daripada membaca dari memori utama atau disk.

data hangat

Data yang jarang diakses. Saat menanyakan jenis data ini, kueri yang cukup lambat biasanya dapat diterima.

fungsi jendela

Fungsi SQL yang melakukan perhitungan pada sekelompok baris yang berhubungan dengan catatan saat ini. Fungsi jendela berguna untuk memproses tugas, seperti menghitung rata-rata bergerak atau mengakses nilai baris berdasarkan posisi relatif dari baris saat ini.

beban kerja

Kumpulan sumber daya dan kode yang memberikan nilai bisnis, seperti aplikasi yang dihadapi pelanggan atau proses backend.

aliran kerja

Grup fungsional dalam proyek migrasi yang bertanggung jawab atas serangkaian tugas tertentu. Setiap alur kerja independen tetapi mendukung alur kerja lain dalam proyek. Misalnya, alur kerja portofolio bertanggung jawab untuk memprioritaskan aplikasi, perencanaan gelombang, dan mengumpulkan metadata migrasi. Alur kerja portofolio mengirimkan aset ini ke alur kerja migrasi, yang kemudian memigrasikan server dan aplikasi.

CACING

Lihat menulis sekali, baca banyak.

WQF

Lihat AWS Kerangka Kualifikasi Beban Kerja.

tulis sekali, baca banyak (WORM)

Model penyimpanan yang menulis data satu kali dan mencegah data dihapus atau dimodifikasi. Pengguna yang berwenang dapat membaca data sebanyak yang diperlukan, tetapi mereka tidak dapat mengubahnya. Infrastruktur penyimpanan data ini dianggap tidak dapat diubah.

Z

eksploitasi zero-day

Serangan, biasanya malware, yang memanfaatkan kerentanan zero-day.

kerentanan zero-day

Cacat atau kerentanan yang tak tanggung-tanggung dalam sistem produksi. Aktor ancaman dapat menggunakan jenis kerentanan ini untuk menyerang sistem. Pengembang sering menyadari kerentanan sebagai akibat dari serangan tersebut.

bisikan zero-shot

Memberikan LLM dengan instruksi untuk melakukan tugas tetapi tidak ada contoh (tembakan) yang dapat membantu membimbingnya. LLM harus menggunakan pengetahuan pra-terlatih untuk menangani tugas. Efektivitas bidikan nol tergantung pada kompleksitas tugas dan kualitas prompt. Lihat juga beberapa bidikan yang diminta.

aplikasi zombie

Aplikasi yang memiliki CPU rata-rata dan penggunaan memori di bawah 5 persen. Dalam proyek migrasi, adalah umum untuk menghentikan aplikasi ini.

Terjemahan disediakan oleh mesin penerjemah. Jika konten terjemahan yang diberikan bertentangan dengan versi bahasa Inggris aslinya, utamakan versi bahasa Inggris.