AWS Prescriptive Guidance

Cloud design patterns, architectures, and implementations

AWS Prescriptive Guidance: Cloud design patterns, architectures, and implementations

Table of Contents

Cloud design patterns, architectures, and implementations

Anitha Deenadayalan, Amazon Web Services (AWS)

September 2023 (document history (p. 51))

This guide provides guidance for implementing commonly used modernization design patterns by using AWS services. An increasing number of modern applications are designed by using microservices architectures to achieve scalability, improve release velocity, reduce the scope of impact for changes, and reduce regression. This leads to improved developer productivity and increased agility, better innovation, and an increased focus on business needs. Microservices architectures also support the use of the best technology for the service and the database, and promote polyglot code and polyglot persistence.

Traditionally, monolithic applications run in a single process, use one data store, and run on servers that scale vertically. In comparison, modern microservice applications are fine-grained, have independent fault domains, run as services across the network, and can use more than one data store depending on the use case. The services scale horizontally, and a single transaction might span multiple databases. Development teams must focus on network communication, polyglot persistence, horizontal scaling, eventual consistency, and transaction handling across the data stores when developing applications by using microservices architectures. Therefore, modernization patterns are critical for solving commonly occurring problems in modern application development, and they help accelerate software delivery.

This guide provides a technical reference for cloud architects, technical leads, application and business owners, and developers who want to choose the right cloud architecture for design patterns based on well-architected best practices. Each pattern discussed in this guide addresses one or more known scenarios in microservices architectures. The guide discusses the issues and considerations associated with each pattern, provides a high-level architectural implementation, and describes the AWS implementation for the pattern. Open-source GitHub samples and workshop links are provided where available.

The guide covers the following patterns:

Targeted business outcomes

By using the patterns discussed in this guide to modernize your applications, you can:

- Design and implement reliable, secure, operationally efficient architectures that are optimized for cost and performance.

- Reduce the cycle time for use cases that require these patterns, so you can focus on organization-specific challenges instead.
- Accelerate development by standardizing pattern implementations by using AWS services.
- Help your developers build modern applications without inheriting technical debt.

Anti-corruption layer pattern

Intent

The anti-corruption layer (ACL) pattern acts as a mediation layer that translates domain model semantics from one system to another system. It translates the model of the upstream bounded context (monolith) into a model that suits the downstream bounded context (microservice) before consuming the communication contract that's established by the upstream team. This pattern might be applicable when the downstream bounded context contains a core subdomain, or the upstream model is an unmodifiable legacy system. It also reduces transformation risk and business disruption by preventing changes to callers when their calls have to be redirected transparently to the target system.

Motivation

During the migration process, when a monolithic application is migrated into microservices, there might be changes in the domain model semantics of the newly migrated service. When the features within the monolith are required to call these microservices, the calls should be routed to the migrated service without requiring any changes to the calling services. The ACL pattern allows the monolith to call the microservices transparently by acting as an adapter or a facade layer that translates the calls into the newer semantics.

Applicability

Consider using this pattern when:

- Your existing monolithic application has to communicate with a function that has been migrated into a microservice, and the migrated service domain model and semantics differ from the original feature.
- Two systems have different semantics and need to exchange data, but it isn't practical to modify one system to be compatible with the other system.
- You want to use a quick and simplified approach to adapt one system to another with minimal impact.
- Your application is communicating with an external system.

Issues and considerations

- **Team dependencies:** When different services in a system are owned by different teams, the new domain model semantics in the migrated services can lead to changes in the calling systems. However, teams might not be able to make these changes in a coordinated way, because they might have other priorities. ACL decouples the callees and translates the calls to match the semantics of the new services, thus avoiding the need for callers to make changes in the current system.
- **Operational overhead:** The ACL pattern requires additional effort to operate and maintain. This work includes integrating ACL with monitoring and alerting tools, the release process, and continuous integration and continuous delivery (CI/CD) processes.
- **Single point of failure:** Any failures in the ACL can make the target service unreachable, causing application issues. To mitigate this issue, you should build in retry capabilities and circuit breakers. See the retry with backoff (p. 23) and circuit breaker (p. 14) patterns to understand more about these options. Setting up appropriate alerts and logging will improve the mean time to resolution (MTTR).

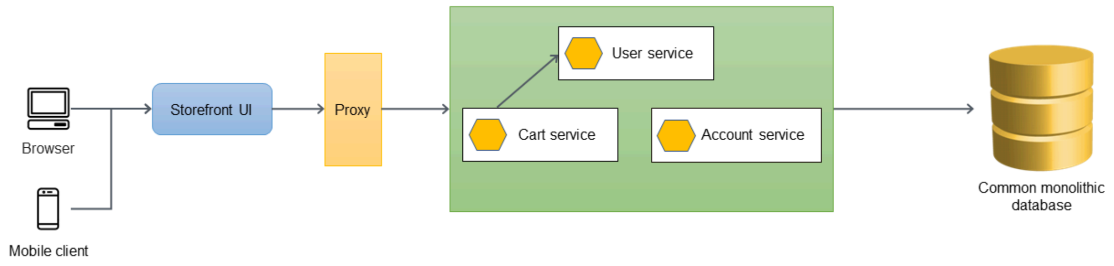
- **Technical debt:** As part of your migration or modernization strategy, consider whether the ACL will be a transient or interim solution, or a long-term solution. If it's an interim solution, you should record the ACL as a technical debt and decommission it after all dependent callers have been migrated.
- **Latency:** The additional layer can introduce latency due to the conversion of requests from one interface to another. We recommend that you define and test performance tolerance in applications that are sensitive to response time before you deploy ACL into production environments.
- **Scaling bottleneck:** In high-load applications where services can scale to peak load, ACL can become a bottleneck and might cause scaling issues. If the target service scales on demand, you should design ACL to scale accordingly.
- **Service-specific or shared implementation:** You can design ACL as a shared object to convert and redirect calls to multiple services or service-specific classes. Take latency, scaling, and failure tolerance into account when you determine the implementation type for ACL.

Implementation

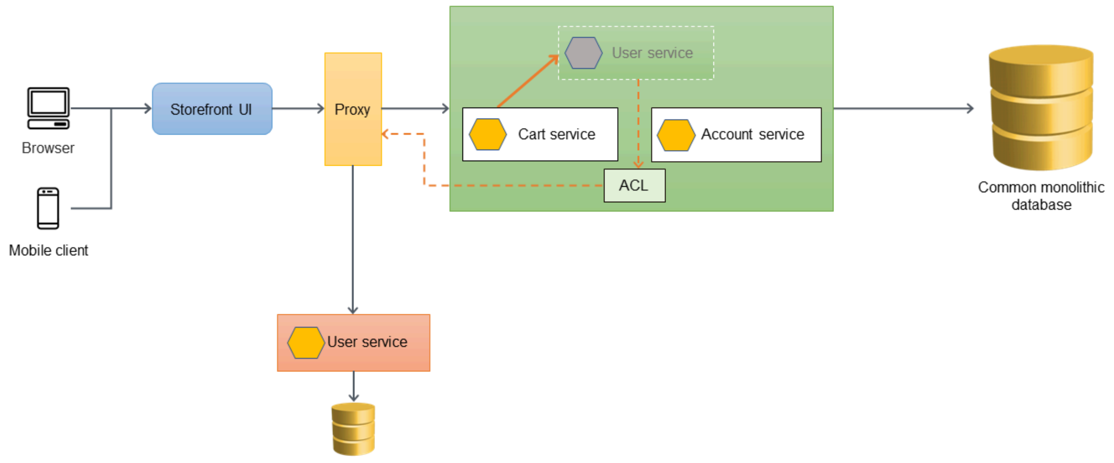
You can implement ACL inside your monolithic application as a class that's specific to the service that's being migrated, or as an independent service. The ACL must be decommissioned after all dependent services have been migrated into the microservices architecture.

High-level architecture

In the following example architecture, a monolithic application has three services: user service, cart service, and account service. The cart service is dependent on the user service, and the application uses a monolithic relational database.

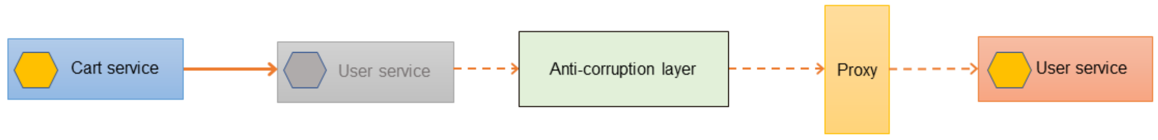


In the following architecture, the user service has been migrated to a new microservice. The cart service calls the user service, but the implementation is no longer available within the monolith. It's also likely that the interface of the newly migrated service won't match its previous interface, when it was inside the monolithic application.



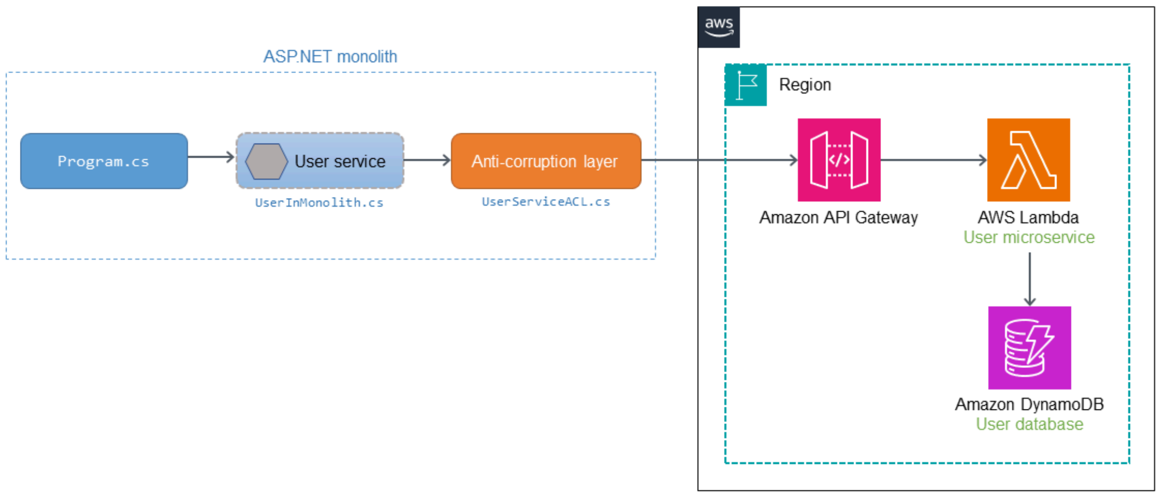
If the cart service has to call the newly migrated user service directly, this will require changes to the cart service and a thorough testing of the monolithic application. This can increase the transformation risk and business disruption. The goal should be to minimize the changes to the existing functionality of the monolithic application.

In this case, we recommend that you introduce an ACL between the old user service and the newly migrated user service. The ACL works as an adapter or a facade that converts the calls into the newer interface. ACL can be implemented inside the monolithic application as a class (for example, `UserServiceFacade` or `UserServiceAdapter`) that's specific to the service that was migrated. The anti-corruption layer must be decommissioned after all dependent services have been migrated into the microservices architecture.



Implementation using AWS services

The following diagram shows how you can implement this ACL example by using AWS services.



The user microservice is migrated out of the ASP.NET monolithic application and deployed as an AWS Lambda function on AWS. Calls to the Lambda function are routed through Amazon API Gateway. ACL is deployed in the monolith to translate the call to adapt to the semantics of the user microservice.

When `Program.cs` calls the user service (`UserInMonolith.cs`) inside the monolith, the call is routed to the ACL (`UserServiceACL.cs`). The ACL translates the call to the new semantics and interface, and calls the microservice through the API Gateway endpoint. The caller (`Program.cs`) isn't aware of the translation and routing that take place in the user service and ACL. Because the caller isn't aware of the code changes, there is less business disruption and reduced transformation risk.

Sample code

The following code snippet provides the changes to the original service and the implementation of `UserServiceACL.cs`. When a request is received, the original user service calls the ACL. The ACL converts the source object to match the interface of the newly migrated service, calls the service, and returns the response to the caller.

```
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 repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/anti-corruption-layer-pattern.

Related content

- Strangler fig pattern (p. 35)
- Circuit breaker pattern (p. 14)
- Retry with backoff pattern (p. 23)

API routing patterns

In agile development environments, autonomous teams (for example squads and tribes) own one or more services that include many microservices. The teams expose these services as APIs to allow their consumers to interact with their group of services and actions.

There are three major methods for exposing HTTP APIs to upstream consumers by using hostnames and paths:

Method	Description	Example
Hostname routing (p. 8)	Expose each service as a hostname.	`billing.api.example.com`
Path routing (p. 9)	Expose each service as a path.	`api.example.com/billing`
Header-based routing (p. 12)	Expose each service as an HTTP header.	`x-example-action: something`

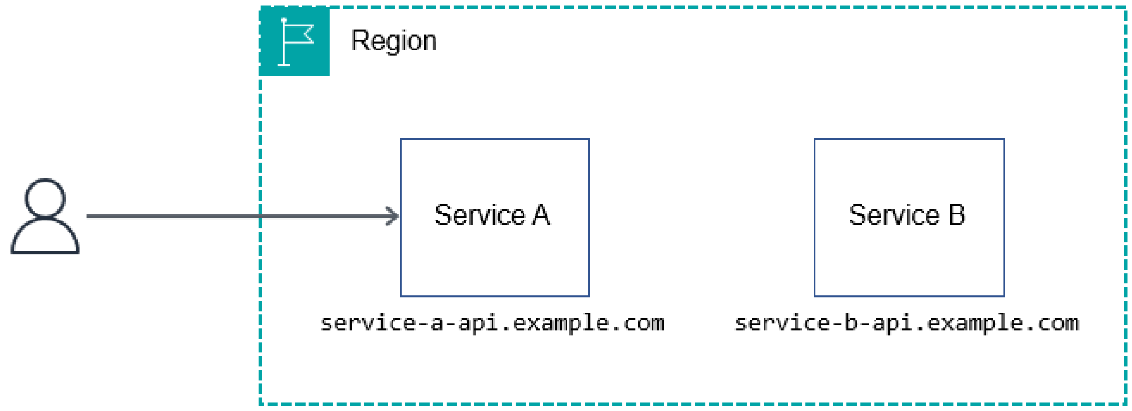
This section outlines typical use cases for these three routing methods and their trade-offs to help you decide which method best fits your requirements and organizational structure.

Hostname routing

Routing by hostname is a mechanism for isolating API services by giving each API its own hostname; for example, `service-a.api.example.com` or `service-a.example.com`.

Typical use case

Routing by using hostnames reduces the amount of friction in releases, because nothing is shared between service teams. Teams are responsible for managing everything from DNS entries to service operations in production.



Pros

Hostname routing is by far the most straightforward and scalable method for HTTP API routing. You can use any relevant AWS service to build an architecture that follows this method—you can create an architecture with Amazon API Gateway, AWS AppSync, Application Load Balancers and Amazon Elastic Compute Cloud (Amazon EC2), or any other HTTP-compliant service.

Teams can use hostname routing to fully own their subdomain. It also makes it easier to isolate, test, and orchestrate deployments for specific AWS Regions or versions; for example, `region.service-a.api.example.com` or `dev.region.service-a.api.example.com`.

Cons

When you use hostname routing, your consumers have to remember different hostnames to interact with each API that you expose. You can mitigate this issue by providing a client SDK.

However, client SDKs come with their own set of challenges. For example, they have to support rolling updates, multiple languages, versioning, communicating breaking changes caused by security issues or bug fixes, documentation, and so on.

Path routing

Routing by paths is the mechanism of grouping multiple or all APIs under the same hostname, and using a request URI to isolate services; for example, `api.example.com/service-a` or `api.example.com/service-b`.

Typical use case

Most teams opt for this method because they want a simple architecture—a developer has to remember only one URL such as `api.example.com` to interact with the HTTP API. API documentation is often easier to digest because it is often kept together instead of being split across different portals or PDFs.

Path-based routing is considered a simple mechanism for sharing an HTTP API. However, it involves operational overhead such as configuration, authorization, integrations, and additional latency due to multiple hops. It also requires mature change management processes to ensure that a misconfiguration doesn't disrupt all services.

On AWS, there are multiple ways to share an API and route effectively to the correct service. The following sections discuss three approaches: HTTP service reverse proxy, API Gateway, and Amazon CloudFront. None of the suggested approaches for unifying API services relies on the downstream services running on AWS. The services could run anywhere without issue or on any technology, as long as they're HTTP-compatible.

HTTP service reverse proxy

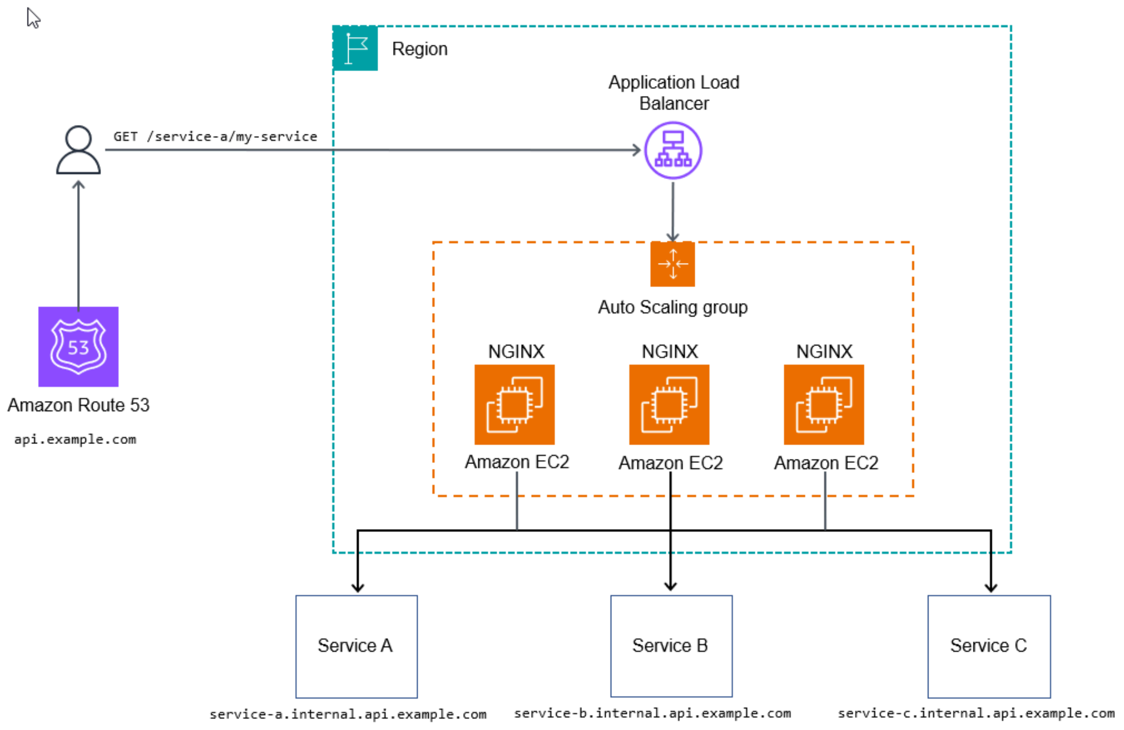
You can use an HTTP server such as NGINX to create dynamic routing configurations. In a Kubernetes architecture, you can also create an ingress rule to match a path to a service. (This guide doesn't cover Kubernetes ingress; see the Kubernetes documentation for more information.)

The following configuration for NGINX dynamically maps an HTTP request of `api.example.com/my-service/` to `my-service.internal.api.example.com`.

```
server {
    listen  80;

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

The following diagram illustrates the HTTP service reverse proxy method.


Region
Application Load Balancer
GET /service-a/my-service
Auto Scaling group
NGINX
NGINX
NGINX
Amazon EC2
Amazon EC2
Amazon EC2
Amazon Route 53
api.example.com
Service A
Service B
Service C
service-a.internal.api.example.com
service-b.internal.api.example.com
service-c.internal.api.example.com

This approach might be sufficient for some use cases that don't use additional configurations to start processing requests, allowing for the downstream API to collect metrics and logs.

To get ready for operational production readiness, you will want to be able to add observability to every level of your stack, add additional configuration, or add scripts to customize your API ingress point to allow for more advanced features such as rate limiting or usage tokens.

Pros

The ultimate aim of the HTTP service reverse proxy method is to create a scalable and manageable approach to unifying APIs into a single domain so it appears coherent to any API consumer. This approach also enables your service teams to deploy and manage their own APIs, with minimal overhead after deployment. AWS managed services for tracing, such as AWS X-Ray or AWS WAF, are still applicable here.

Cons

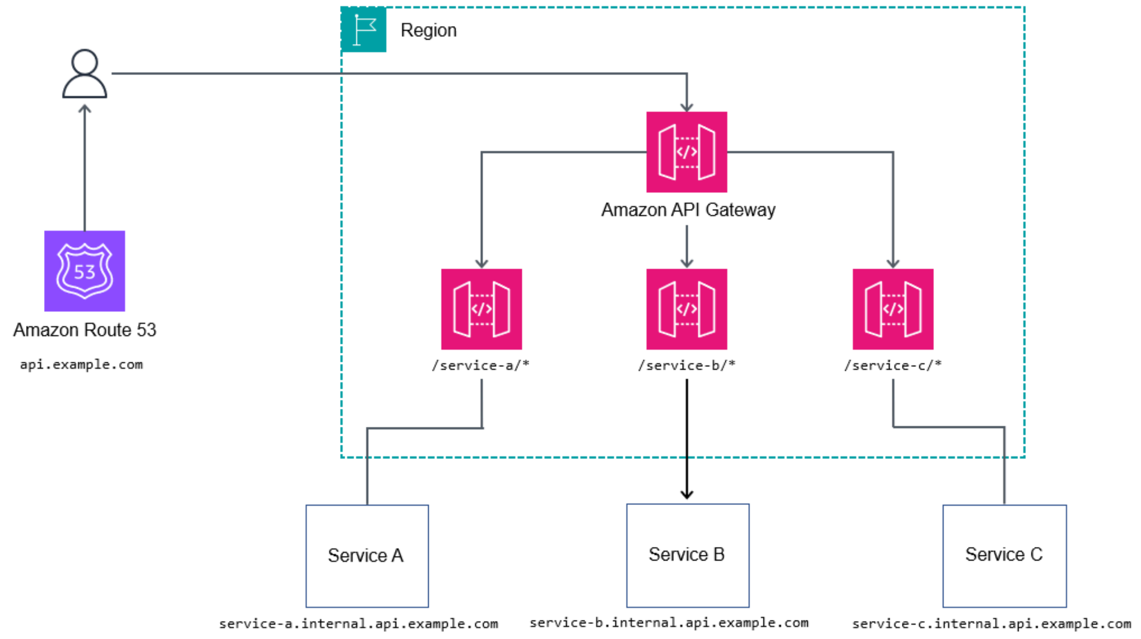
The major downside of this approach is the extensive testing and management of infrastructure components that are required, although this might not be an issue if you have site reliability engineering (SRE) teams in place.

There is a cost tipping point with this method. At low to medium volumes, it is more expensive than some of the other methods discussed in this guide. At high volumes, it is very cost-effective (around 100K transactions per second or better).

API Gateway

The Amazon API Gateway service (REST APIs and HTTP APIs) can route traffic in a way that's similar to the HTTP service reverse proxy method. Using an API gateway in HTTP proxy mode provides a simple way to wrap many services into an entry point to the top-level subdomain `api.example.com`, and then proxy requests to the nested service; for example, `billing.internal.api.example.com`.

You probably don't want to get too granular by mapping every path in every service in the root or core API gateway. Instead, opt for wildcard paths such as `/billing/*` to forward requests to the billing service. By not mapping every path in the root or core API gateway, you gain more flexibility over your APIs, because you don't have to update the root API gateway with every API change.



Pros

For control over more complex workflows, such as changing request attributes, REST APIs expose the Apache Velocity Template Language (VTL) to allow you to modify the request and response. REST APIs can provide additional benefits such as these:

- Auth N/Z with AWS Identity and Access Management (IAM), Amazon Cognito, or AWS Lambda authorizers
- AWS X-Ray for tracing
- Integration with AWS WAF
- Basic rate limiting
- Usage tokens for bucketing consumers into different tiers (see Throttle API requests for better throughput in the API Gateway documentation)

Cons

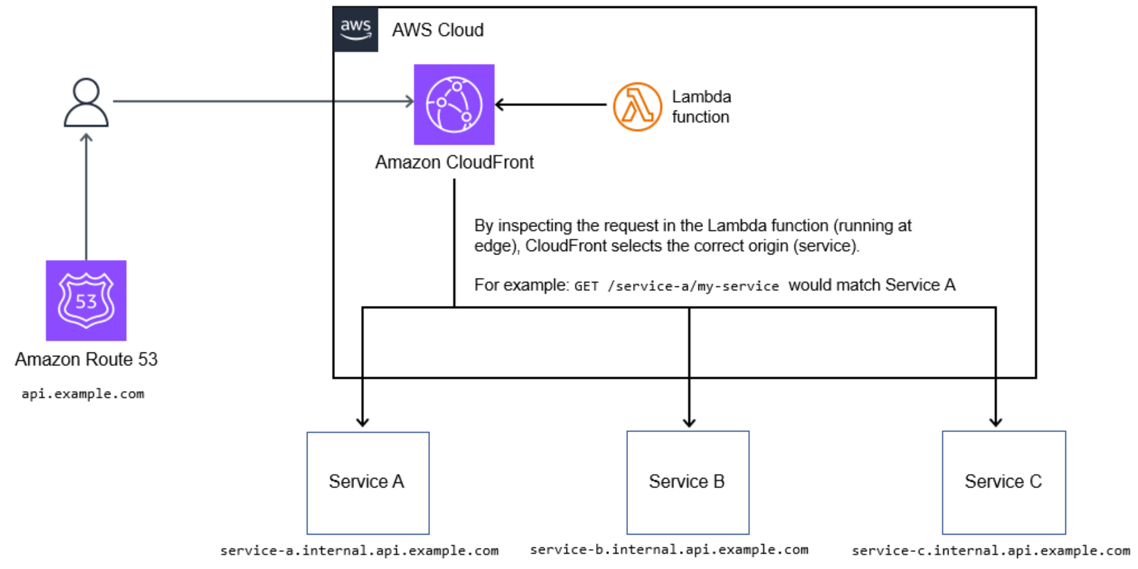
At high volumes, cost might be an issue for some users.

CloudFront

You can use the dynamic origin selection feature in Amazon CloudFront to conditionally select an origin (a service) to forward the request. You can use this feature to route a number of services through a single hostname such as `api.example.com`.

Typical use case

The routing logic lives as code within the Lambda@Edge function, so it supports highly customizable routing mechanisms such as A/B testing, canary releases, feature flagging, and path rewriting. This is illustrated in the following diagram.


AWS Cloud
Amazon CloudFront
Lambda function
Amazon Route 53
api.example.com
By inspecting the request in the Lambda function (running at edge), CloudFront selects the correct origin (service).
For example: GET /service-a/my-service would match Service A
Service A
Service B
Service C
service-a.internal.api.example.com
service-b.internal.api.example.com
service-c.internal.api.example.com

Pros

If you require caching API responses, this method is good way to unify a collection of services behind a single endpoint. It is a cost-effective method to unify collections of APIs.

Also, CloudFront supports field-level encryption as well as integration with AWS WAF for basic rate limiting and basic ACLs.

Cons

This method supports a maximum of 250 origins (services) that can be unified. This limit is sufficient for most deployments, but it might cause issues with a large number of APIs as you grow your portfolio of services.

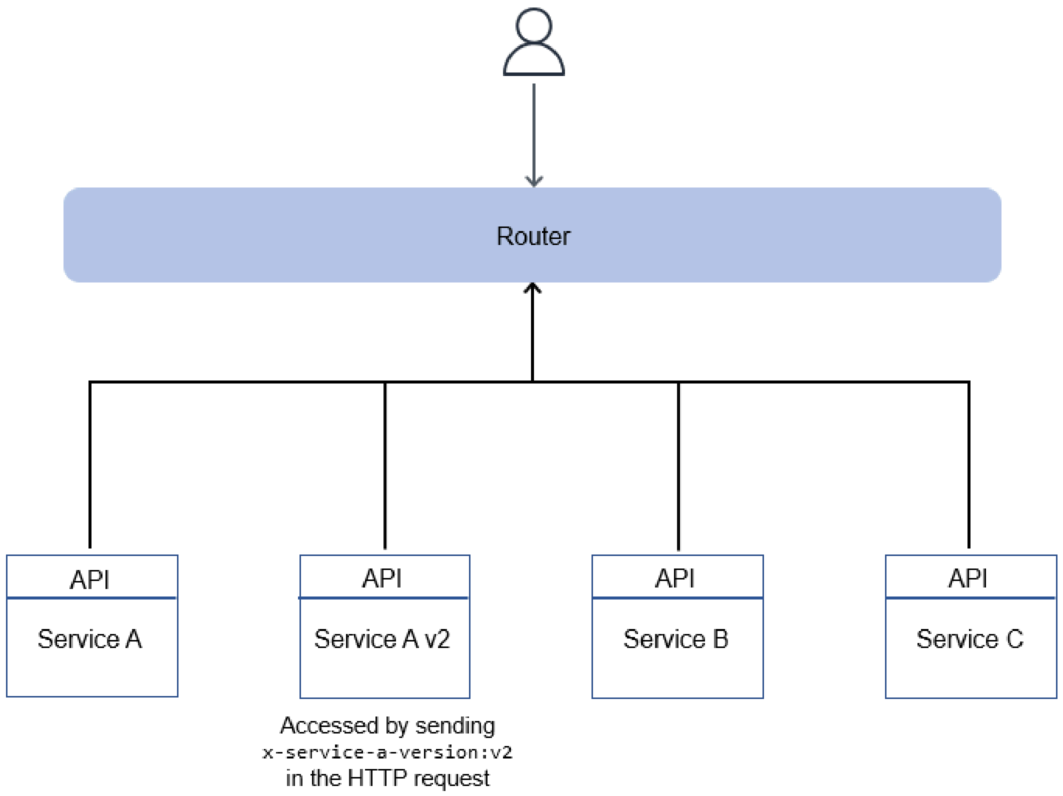
Updating Lambda@Edge functions currently takes a few minutes. CloudFront also takes up to 30 minutes to complete propagating changes to all points of presence. This ultimately blocks further updates until they complete.

HTTP header routing

Header-based routing enables you to target the correct service for each request by specifying an HTTP header in the HTTP request. For example, sending the header `x-service-a-action: get-thing` would enable you to `get thing` from `Service A`. The path of the request is still important, because it offers guidance on which resource you're trying to work on.

In addition to using HTTP header routing for actions, you can use it as a mechanism for version routing, enabling feature flags, A/B tests, or similar needs. In reality, you will likely use header routing with one of the other routing methods to create robust APIs.

The architecture for HTTP header routing typically has a thin routing layer in front of microservices that routes to the correct service and returns a response, as illustrated in the following diagram. This routing layer could cover all services or just a few services to enable an operation such as version-based routing.



Pros

Configuration changes require minimal effort and can be automated easily. This method is also flexible and supports creative ways to expose only specific operations you would want from a service.

Cons

As with the hostname routing method, HTTP header routing assumes that you have full control over the client and can manipulate custom HTTP headers. Proxies, content delivery networks (CDNs), and load balancers can limit the header size. Although this is unlikely to be a concern, it could be an issue depending on how many headers and cookies you add.

Circuit breaker pattern

Intent

The circuit breaker pattern can prevent a caller service from retrying a call to another service (*callee*) when the call has previously caused repeated timeouts or failures. The pattern is also used to detect when the callee service is functional again.

Motivation

When multiple microservices collaborate to handle requests, one or more services might become unavailable or exhibit high latency. When complex applications use microservices, an outage in one microservice can lead to application failure. Microservices communicate through remote procedure calls, and transient errors could occur in network connectivity, causing failures. (The transient errors can be handled by using the retry with backoff (p. 23) pattern.) During synchronous execution, the cascading of timeouts or failures can cause a poor user experience.

However, in some situations, the failures could take longer to resolve—for example, when the callee service is down or a database contention results in timeouts. In such cases, if the calling service retries the calls repeatedly, these retries might result in network contention and database thread pool consumption. Additionally, if multiple users are retrying the application repeatedly, this will exacerbate the problem and can cause performance degradation in the entire application.

The circuit breaker pattern was popularized by Michael Nygard in his book, *Release It* (Nygard 2018). This design pattern can prevent a caller service from retrying a service call that has previously caused repeated timeouts or failures. It can also detect when the callee service is functional again.

Circuit breaker objects work like electrical circuit breakers that automatically interrupt the current when there is an abnormality in the circuit. Electrical circuit breakers shut off, or trip, the flow of the current when there is a fault. Similarly, the circuit breaker object is situated between the caller and the callee service, and trips if the callee is unavailable.

The fallacies of distributed computing are a set of assertions made by Peter Deutsch and others at Sun Microsystems. They say that programmers who are new to distributed applications invariably make false assumptions. The network reliability, zero-latency expectations, and bandwidth limitations result in software applications written with minimal error handling for network errors.

During a network outage, applications might indefinitely wait for a reply and continually consume application resources. Failure to retry the operations when the network becomes available can also lead to application degradation. If API calls to a database or an external service time out because of network issues, repeated calls with no circuit breaker can affect cost and performance.

Applicability

Use this pattern when:

- The caller service makes a call that is most likely going to fail.
- A high latency exhibited by the callee service (for example, when database connections are slow) causes timeouts to the callee service.
- The caller service makes a synchronous call, but the callee service isn't available or exhibits high latency.

Issues and considerations

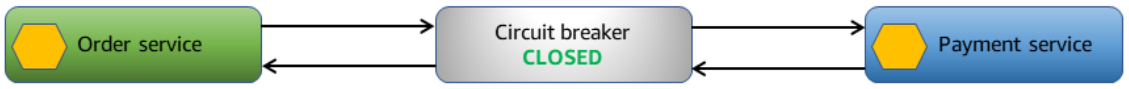
- **Service agnostic implementation:** To prevent code bloat, we recommend that you implement the circuit breaker object in a microservice-agnostic and API-driven way.
- **Circuit closure by callee:** When the callee recovers from the performance issue or failure, they can update the circuit status to `CLOSED`. This is an extension of the circuit breaker pattern and can be implemented if your recovery time objective (RTO) requires it.
- **Multithreaded calls:** The expiration timeout value is defined as the period of time the circuit remains tripped before calls are routed again to check for service availability. When the callee service is called in multiple threads, the first call that failed defines the expiration timeout value. Your implementation should ensure that subsequent calls do not move the expiration timeout endlessly.
- **Force open or close the circuit:** System administrators should have the ability to open or close a circuit. This can be done by updating the expiration timeout value in the database table.
- **Observability:** The application should have logging set up to identify the calls that fail when the circuit breaker is open.

Implementation

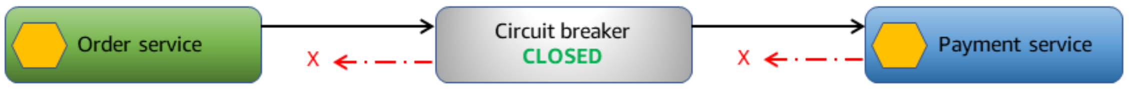
High-level architecture

In the following example, the caller is the order service and the callee is the payment service.

When there are no failures, the order service routes all calls to the payment service by the circuit breaker, as the following diagram shows.

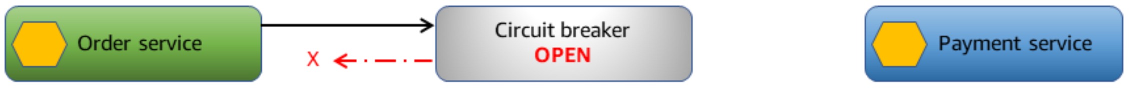


If the payment service times out, the circuit breaker can detect the timeout and track the failure.



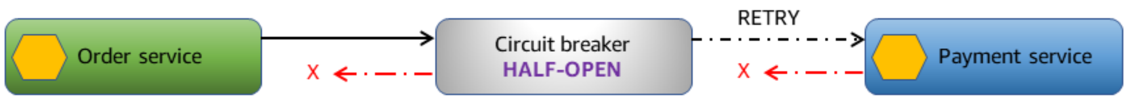
Circuit breaker with payment service failure

If the timeouts exceed a specified threshold, the application opens the circuit. When the circuit is open, the circuit breaker object doesn't route the calls to the payment service. It returns an immediate failure when the order service calls the payment service.



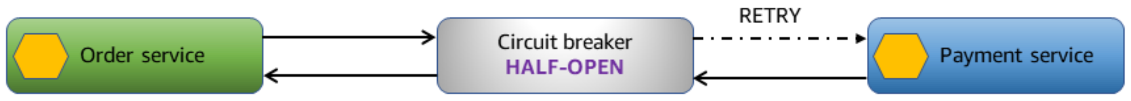
Circuit breaker stops routing to payment service

The circuit breaker object periodically tries to see if the calls to the payment service are successful.



Circuit breaker periodically retries payment service

When the call to payment service succeeds, the circuit is closed, and all further calls are routed to the payment service again.



Circuit breaker with working payment service

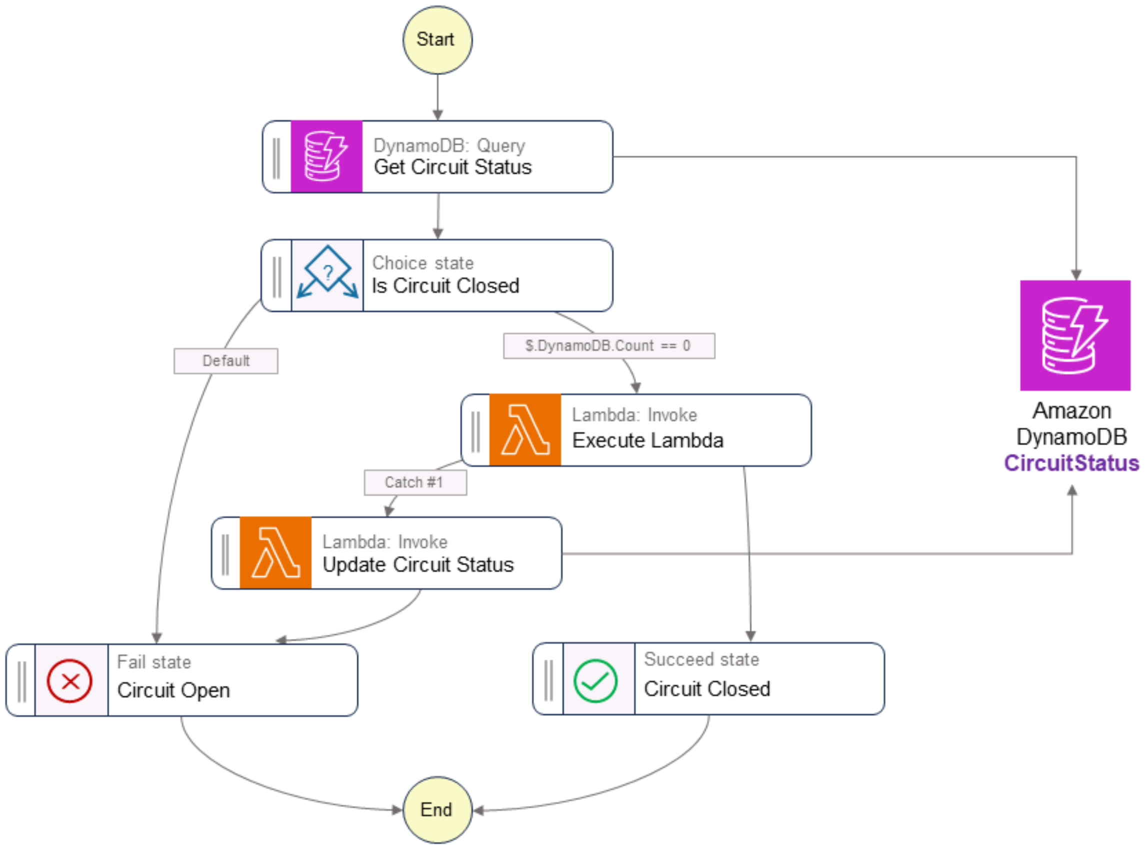
Implementation using AWS services

The sample solution uses express workflows in [AWS Step Functions](https://aws.amazon.com/step-functions/) to implement the circuit breaker pattern. The Step Functions state machine lets you configure the retry capabilities and decision-based control flow required for the pattern implementation.

The solution also uses an [Amazon DynamoDB](https://aws.amazon.com/dynamodb/) table as the data store to track the circuit status. This can be replaced with an in-memory datastore such as [Amazon ElastiCache for Redis](https://aws.amazon.com/elasticache/redis/) for better performance.

When a service wants to call another service, it starts the workflow with the name of the callee service. The workflow gets the circuit breaker status from the DynamoDB `CircuitStatus` table, which stores the currently degraded services. If `CircuitStatus` contains an unexpired record for the callee, the circuit is open. The Step Functions workflow returns an immediate failure and exits with a `FAIL` state.

If the `CircuitStatus` table doesn't contain a record for the callee or contains an expired record, the service is operational. The `ExecuteLambda` step in the state machine definition calls the Lambda function that's sent through a parameter value. If the call succeeds, the Step Functions workflow exits with a `SUCCESS` state.



If the service call fails or a timeout occurs, the application retries with exponential backoff for a defined number of times. If the service call fails after the retries, the workflow inserts a record in the `CircuitStatus` table for the service with the an `ExpiryTimeStamp`, and the workflow exits with a FAIL state. Subsequent calls to the same service return an immediate failure as long as the circuit breaker is open. The Get Circuit Status step in the state machine definition checks the service availability based on the ExpiryTimeStamp value. Expired items are deleted from the CircuitStatus table by using the DynamoDB time to live (TTL) feature.

Sample code

The following code uses the GetCircuitStatus Lambda function to check the circuit breaker status.

```
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 = "";
}
```

The following code shows the Amazon States Language statements in the Step Functions workflow.

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

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/circuit-breaker-netcore-blog.

Blog references

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

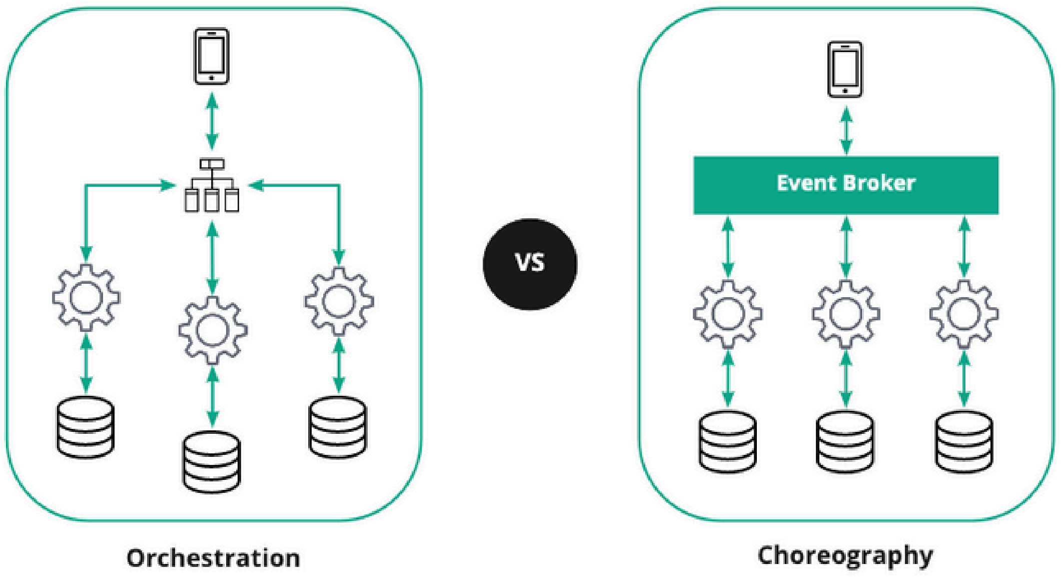
Related content

- Strangler fig pattern (p. 35)

- Retry with backoff pattern (p. 23)
- AWS App Mesh circuit breaker capabilities

Orchestration and choreography patterns

Developers can use microservices architectures to build small, lightweight, decoupled services that are easier to develop and maintain. To form a complete platform, each microservice has to interact with other parts of the system. Orchestration and choreography are two patterns that allow interactions among microservices.



In orchestration (p. 19), a controller (*orchestrator*) handles the flow of interaction between services. Each service request takes place according to a specific order and condition.

In choreography (p. 21), all services work independently and interact only through shared events and loose coupling. Services subscribe only to events that are relevant to them and perform a specific task when they receive a notification.

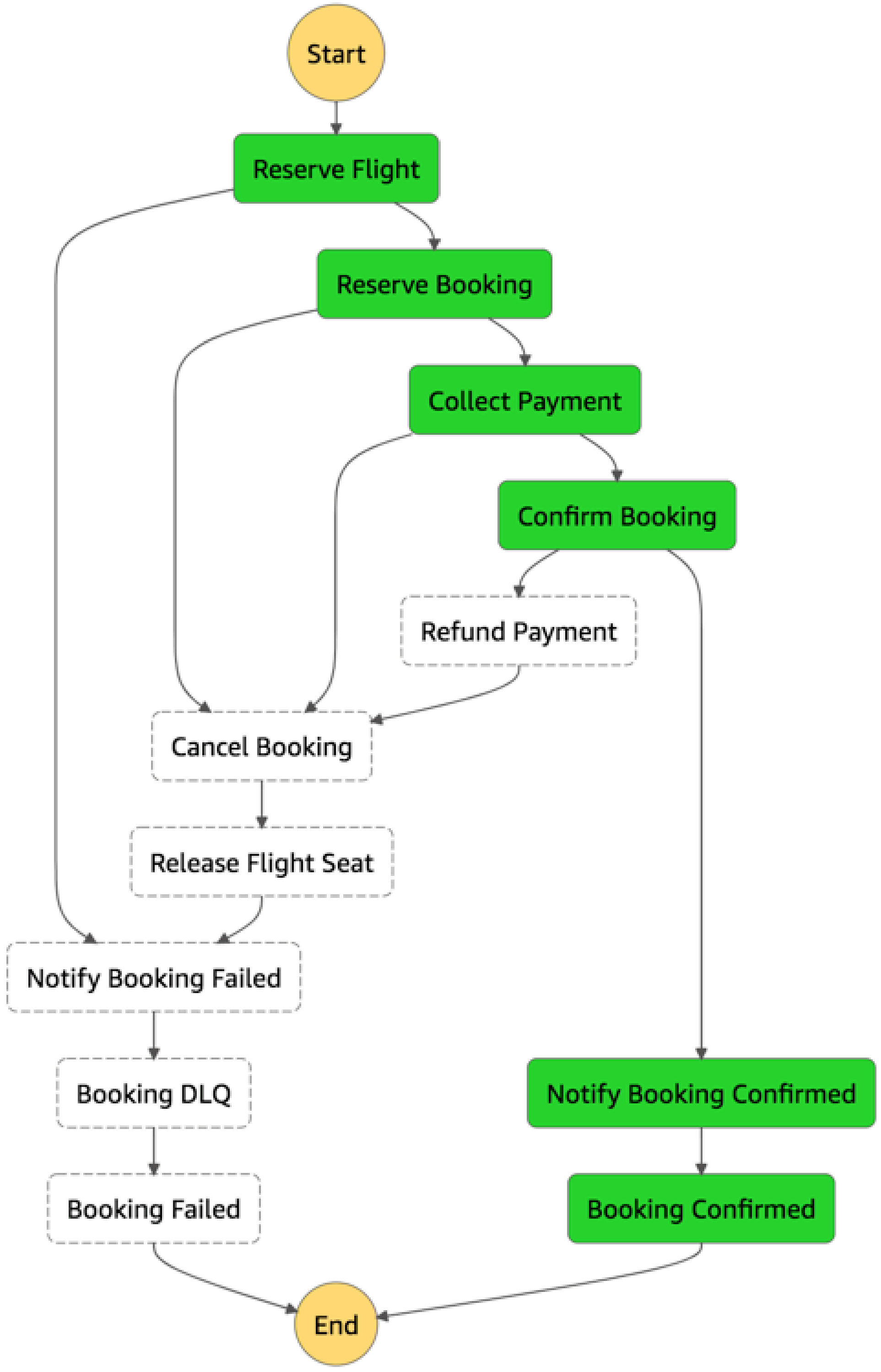
Note
In the context of event-driven architectures, the most popular pattern is choreography. However, depending on your requirements, it might not be the best option.

Orchestration

To understand the orchestration pattern, consider the origin of the pattern name: an orchestra. In an orchestra, each musician is an expert on their instrument, and they wait on the orchestra conductor to give them instructions on when to start playing. An orchestra that doesn't have a conductor can't perform and achieve their objective, which is to play music.

Service orchestration follows a similar pattern where the orchestrator decides on the flow of execution depending on configuration and (potentially) the outcome or response of other services.

The following diagram shows an orchestration pattern implementation that uses AWS Step Functions, from the AWS Serverless Airline Booking sample.



The path follows the steps in *green* if the booking process goes as expected. If something fails in the process, the orchestrator follows an alternative path (using the steps with dashed borders) that notifies the user and the system that something is wrong.

Pros

- Business logic and flow are easy to understand.
- Business logic and flow are source-controlled.
- End-to-end monitoring and reporting are simplified because the orchestrator is implemented by a single service; that is, Step Functions.

  Note
  Step Functions offers built-in flow and execution visualization and auditing.
- It's easy to implement timeout and retry patterns.

Cons

- This pattern follows a request/response pattern. If a service is taking time to reply, the entire system is impacted.
- Services are tightly coupled because they are dependent on a response from another service.
- The orchestrator is a single point of failure. If it isn't carefully designed, you will end up with a monolith.
- Updating the orchestrator or orchestrator logic might introduce downtime.

Choreography

In choreography, as in orchestration, each service performs a specialized task. However, tasks are performed asynchronously because they don't have direct dependencies on specific services. Instead, each service subscribes to certain events and acts when it receives notifications for those events.

This provides better decoupling and resiliency in the system, because each service is an individual unit and can be scaled and updated without affecting other components.

Pros

- Components can be changed independently.
- Components can be scaled independently.
- This pattern enables more agile development than the orchestration pattern.
- There is no single point of failure.
- This pattern enables easier integration with external systems, compared with the orchestration pattern.
- Events can be used for auditing and troubleshooting purposes because they reflect every change in the system.

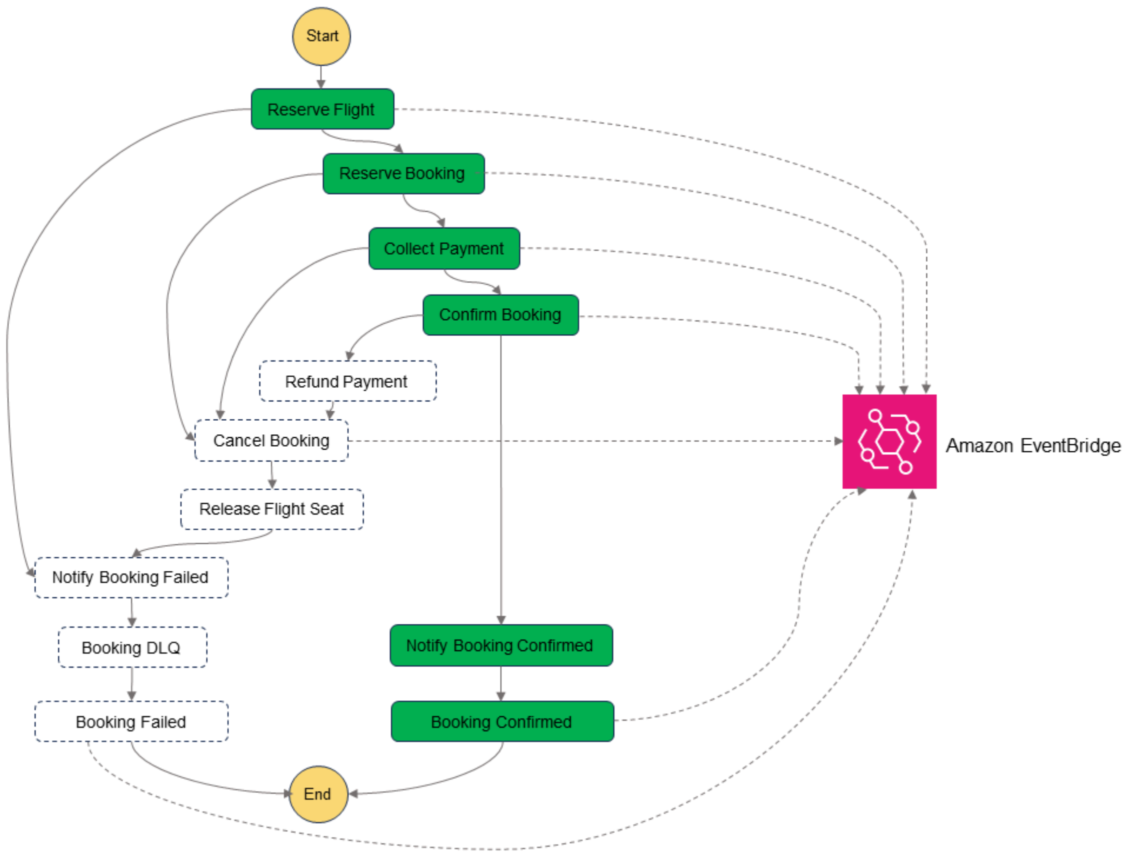
Cons

- It might be difficult to capture full business logic from the implementation because the flow isn't explicitly modelled.
- End-to-end monitoring and reporting are more difficult to achieve compared with the orchestration pattern.
- It's more difficult to implement timeouts, retries, and other resiliency patterns globally, compared with orchestration. This pattern has to be implemented on individual components.

Using orchestration and choreography together

Orchestration and choreography are not necessarily mutually exclusive and can work together. (For example, Step Functions can send relevant state changes to Amazon EventBridge for integration with external services.)

The following diagram shows a possible implementation of the orchestration diagram shown previously. This version implements both patterns—events are sent to the event broker so that other services can subscribe to them.



Retry with backoff pattern

Intent

The retry with backoff pattern improves application stability by transparently retrying operations that fail due to transient errors.

Motivation

In distributed architectures, transient errors might be caused by service throttling, temporary loss of network connectivity, or temporary service unavailability. Automatically retrying operations that fail because of these transient errors improves the user experience and application resilience. However, frequent retries can overload network bandwidth and cause contention. Exponential backoff is a technique where operations are retried by increasing wait times for a specified number of retry attempts.

Applicability

Use the retry with backoff pattern when:

- Your services frequently throttle the request to prevent overload, resulting in a *429 Too many requests* exception to the calling process.
- The network is an unseen participant in distributed architectures, and temporary network issues result in failures.
- The service being called is temporarily unavailable, causing failures. Frequent retries might cause service degradation unless you introduce a backoff timeout by using this pattern.

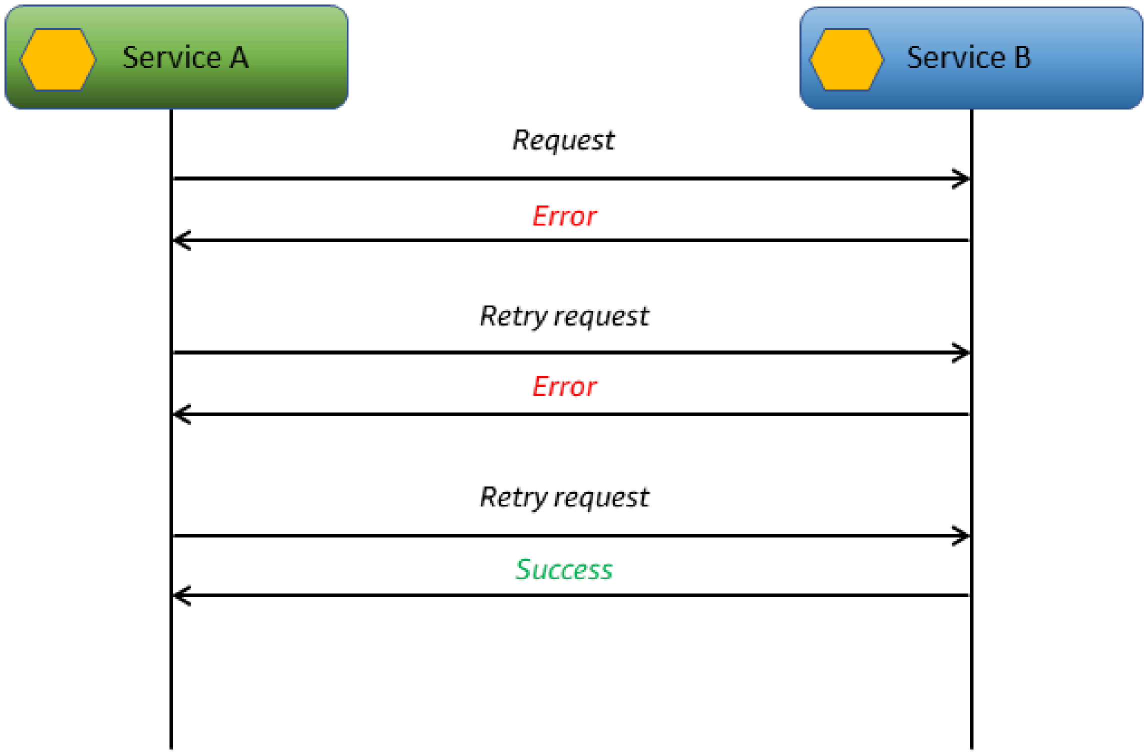
Issues and considerations

- **Idempotency**: If multiple calls to the method have the same effect as a single call on the system state, the operation is considered idempotent. Operations should be idempotent when you use the retry with backoff pattern. Otherwise, partial updates might corrupt the system state.
- **Network bandwidth**: Service degradation can occur if too many retries occupy network bandwidth, leading to slow response times.
- **Fail fast scenarios**: For non-transient errors, if you can determine the cause of the failure, it is more efficient to fail fast by using the circuit breaker pattern.
- **Backoff rate**: Introducing exponential backoff can have an impact on the service timeout, resulting in longer wait times for the end user.

Implementation

High-level architecture

The following diagram illustrates how Service A can retry the calls to Service B until a successful response is returned. If Service B doesn't return a successful response after a few tries, Service A can stop retrying and return a failure to its caller.

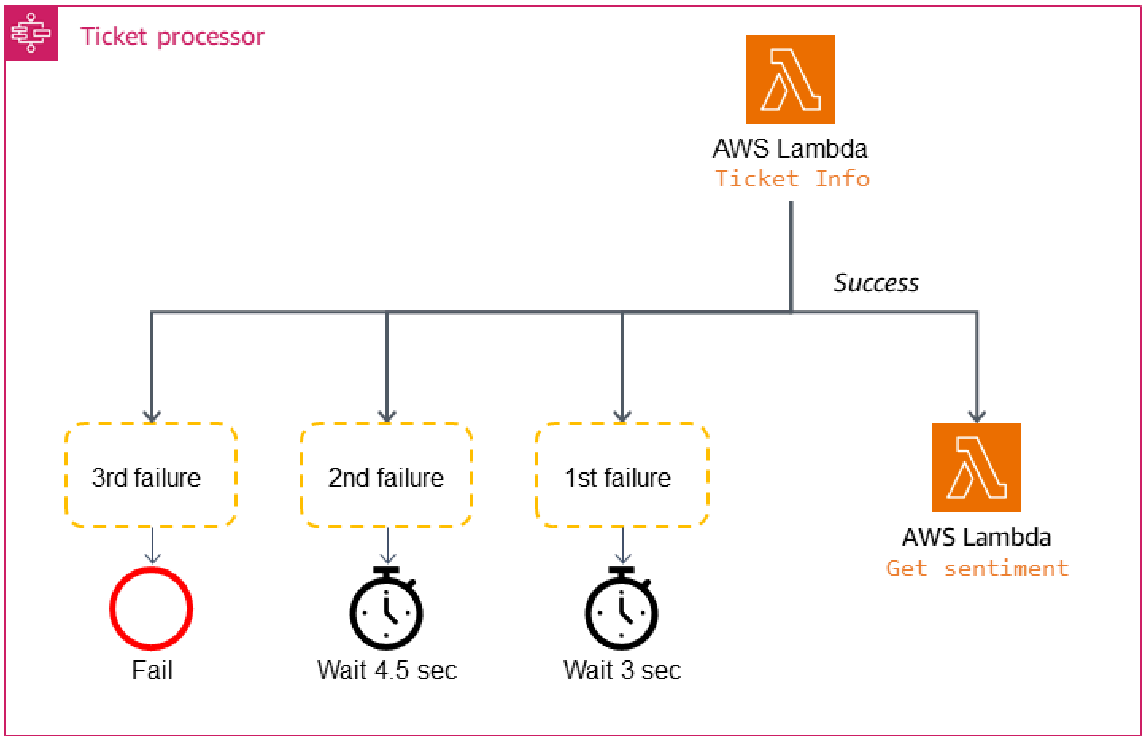


Implementation using AWS services

The following diagram shows a ticket processing workflow on a customer support platform. Tickets from unhappy customers are expedited by automatically escalating the ticket priority. The `Ticket info` Lambda function extracts the ticket details and calls the `Get sentiment` Lambda function. The `Get sentiment` Lambda function checks the customer sentiments by passing the description to [Amazon Comprehend](https://aws.amazon.com/comprehend/) (not shown).

If the call to the `Get sentiment` Lambda function fails, the workflow retries the operation three times. AWS Step Functions allows exponential backoff by letting you configure the backoff value.

In this example, a maximum of three retries are configured with an increase multiplier of 1.5 seconds. If the first retry occurs after 3 seconds, the second retry occurs after 3 x 1.5 seconds = 4.5 seconds, and the third retry occurs after 4.5 x 1.5 seconds = 6.75 seconds. If the third retry is unsuccessful, the workflow fails. The backoff logic doesn't require any custom code—it's provided as a configuration by AWS Step Functions.



Sample code

The following code shows the implementation of the retry with backoff pattern.

```
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 repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/retry-with-backoff.

Related content

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

Saga orchestration

Intent

The saga orchestration pattern helps preserve data integrity in distributed transactions that span multiple services. In a distributed transaction, multiple services can be called before a transaction is completed. When the services store data in different data stores, it can be challenging to maintain data consistency across these data stores.

Motivation

A *transaction* is a single unit of work that might involve multiple steps, where all steps are completely executed or no step is executed, resulting in a data store that retains its consistent state. The terms *atomicity, consistency, isolation, and durability (ACID)* define the properties of a transaction. Relational databases provide ACID transactions to maintain data consistency.

To maintain consistency in a transaction, relational databases use the two-phase commit (2PC) method. This consists of a *prepare phase* and a *commit phase*.

- In the prepare phase, the coordinating process requests the transaction's participating processes (participants) to promise to either commit or roll back the transaction.
- In the commit phase, the coordinating process requests the participants to commit the transaction. If the participants cannot agree to commit in the prepare phase, the transaction is rolled back.

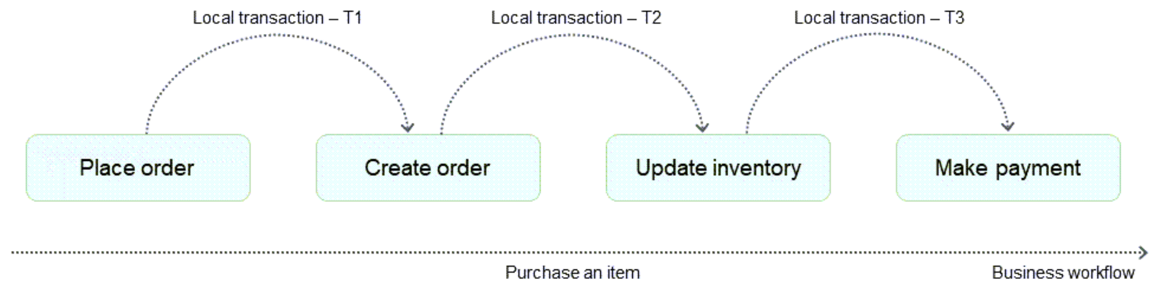
In distributed systems that follow a database-per-service design pattern, the two-phase commit is not an option. This is because each transaction is distributed across various databases, and there is no single controller that can coordinate a process that's similar to the two-phase commit in relational data stores. In this case, one solution is to use the saga orchestration pattern.

What is a saga?

A *saga* consists of a sequence of local transactions. Each local transaction in a saga updates the database and triggers the next local transaction. If a transaction fails, the saga runs compensating transactions to revert the database changes made by the previous transactions.

This sequence of local transactions helps achieve a business workflow by using continuation and compensation principles. The *continuation principle* decides the forward recovery of the workflow, whereas the *compensation principle* decides the backward recovery. If the update fails at any step in the transaction, the saga publishes an event for either continuation (to retry the transaction) or compensation (to go back to the previous data state). This ensures that data integrity is maintained and is consistent across the data stores.

For example, when a user purchases a book from an online retailer, the process consists of a sequence of transactions—such as order creation, inventory update, payment, and shipping—that represents a business workflow. In order to complete this workflow, the distributed architecture issues a sequence of local transactions to create an order in the order database, update the inventory database, and update the payment database. When the process is successful, these transactions are invoked sequentially to complete the business workflow. However, if any of these local transactions fails, the system should be able to decide on an appropriate next step—that is, either a forward recovery or a backward recovery.



The following two scenarios help determine whether the next step is forward recovery or backward recovery:

- Platform-level failure, where something goes wrong with the underlying infrastructure and causes the transaction to fail. In this case, the saga pattern can perform a forward recovery by retrying the local transaction and continuing the business process.
- Application-level failure, where the payment service fails because of an invalid payment. In this case, the saga pattern can perform a backward recovery by issuing a compensatory transaction to update the inventory and the order databases, and reinstate their previous state.

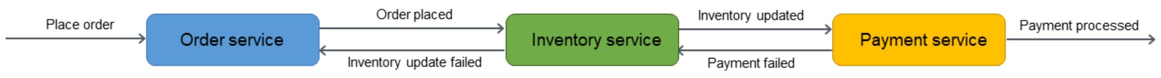
The saga pattern handles the business workflow and ensures that a desirable end state is reached through forward recovery. In case of failures, it reverts the local transactions by using backward recovery to avoid data consistency issues.

The saga pattern has two variants: choreography and orchestration. This guide discusses the saga choreography pattern briefly, and then provides a deep dive into the saga orchestration pattern.

Saga choreography

The saga choreography pattern depends on the events published by the microservices. The saga participants (microservices) subscribe to the events and act based on the event triggers. For example, the order service in the following diagram emits an `OrderPlaced` event. The inventory service subscribes to that event and updates the inventory when the `OrderPlaced` event is emitted. Similarly, the participant services act based on the context of the emitted event.

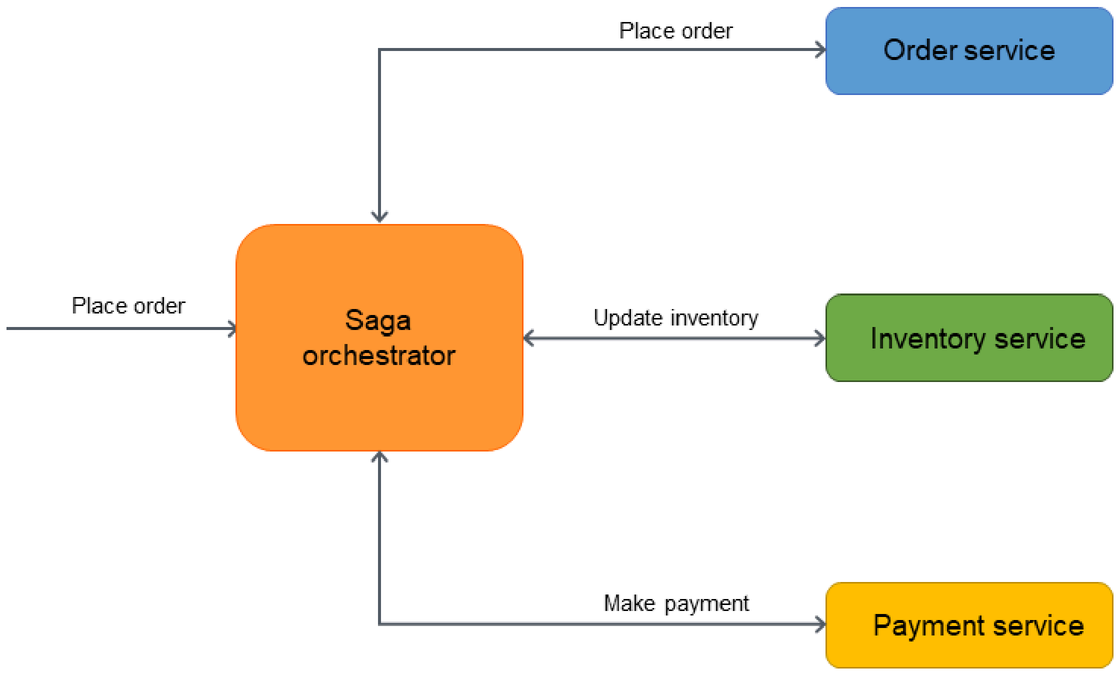
The saga choreography pattern is suitable when there are only a few participants in the saga, and you need a simple implementation with no single point of failure. When more participants are added, it becomes harder to track the dependencies between the participants by using this pattern.



Saga orchestration

The saga orchestration pattern has a central coordinator called an *orchestrator*. The saga orchestrator manages and coordinates the entire transaction lifecycle. It is aware of the series of steps to be performed to complete the transaction. To run a step, it sends a message to the participant microservice to perform the operation. The participant microservice completes the operation and sends a message back to the orchestrator. Based on the message it receives, the orchestrator decides which microservice to run next in the transaction.

The saga orchestration pattern is suitable when there are many participants, and loose coupling is required between saga participants. The orchestrator encapsulates the complexity in the logic by making the participants loosely coupled. However, the orchestrator can become a single point of failure because it controls the entire workflow.



Applicability

Use the saga orchestration pattern when:

- Your system requires data integrity and consistency in distributed transactions that span multiple data stores.
- The data store doesn't provide 2PC to provide ACID transactions, and implementing 2PC within the application boundaries is a complex task.
- You have NoSQL databases, which do not provide ACID transactions, and you need to update multiple tables within a single transaction.

Issues and considerations

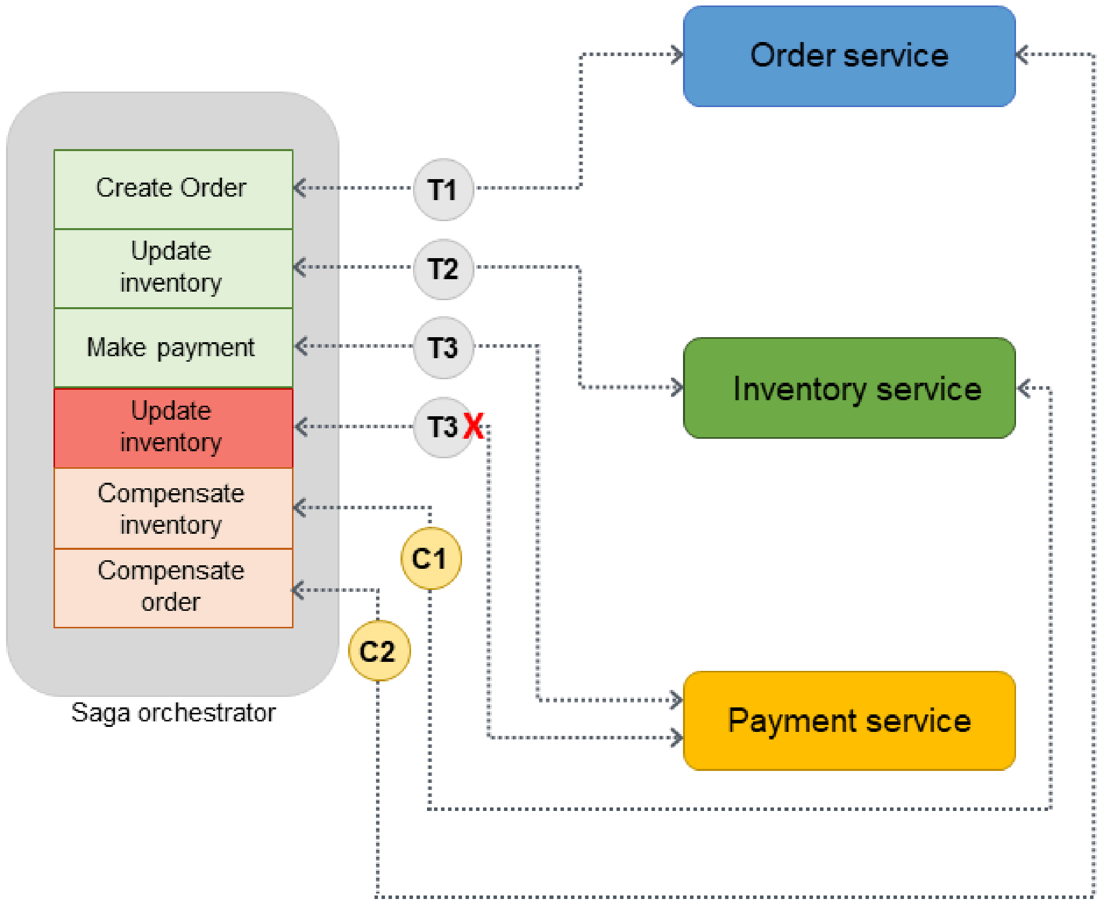
- **Complexity**: Compensatory transactions and retries add complexities to the application code, which can result in maintenance overhead.
- **Eventual consistency**: The sequential processing of local transactions results in eventual consistency, which can be a challenge in systems that require strong consistency. You can address this issue by setting your business teams' expectations for the consistency model or by switching to a data store that provides strong consistency.
- **Idempotency**: Saga participants need to be idempotent to allow repeated execution in case of transient failures caused by unexpected crashes and orchestrator failures.
- **Transaction isolation**: Saga lacks transaction isolation. Concurrent orchestration of transactions can lead to stale data. We recommend using semantic locking to handle such scenarios.
- **Observability**: Observability refers to detailed logging and tracing to troubleshoot issues in the execution and orchestration process. This becomes important when the number of saga participants increases, resulting in complexities in debugging.

- **Latency issues**: Compensatory transactions can add latency to the overall response time when the saga consists of several steps. Avoid synchronous calls in such cases.
- **Single point of failure**: The orchestrator can become a single point of failure because it coordinates the entire transaction. In some cases, the saga choreography pattern is preferred because of this issue.

Implementation

High-level architecture

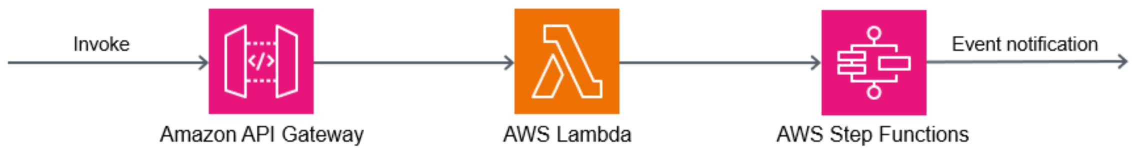
In the following architecture diagram, the saga orchestrator has three participants: the order service, the inventory service, and the payment service. Three steps are required to complete the transaction: T1, T2, and T3. The saga orchestrator is aware of the steps and runs them in the required order. When step T3 fails (payment failure), the orchestrator runs the compensatory transactions C1 and C2 to restore the data to the initial state.



You can use AWS Step Functions to implement saga orchestration when the transaction is distributed across multiple databases.

Implementation using AWS services

The sample solution uses the standard workflow in Step Functions to implement the saga orchestration pattern.



When a customer calls the API, the Lambda function is invoked, and preprocessing occurs in the Lambda function. The function starts the Step Functions workflow to start processing the distributed transaction. If preprocessing isn't required, you can initiate the Step Functions workflow directly from API Gateway without using the Lambda function.

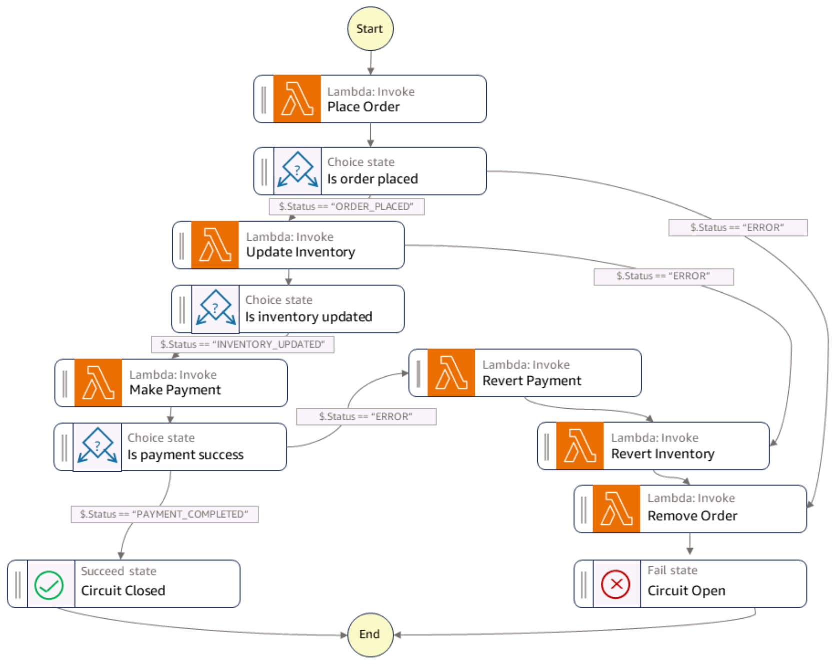
The use of Step Functions mitigates the single point of failure issue, which is inherent in the implementation of the saga orchestration pattern. Step Functions has built-in fault tolerance and maintains service capacity across multiple Availability Zones in each AWS Region to protect applications against individual machine or data center failures. This helps ensure high availability for both the service itself and for the application workflow it operates.

The Step Functions workflow

The Step Functions state machine allows you to configure the decision-based control flow requirements for the pattern implementation. The Step Functions workflow calls the individual services for order placement, inventory update, and payment processing to complete the transaction and sends an event notification for further processing. The Step Functions workflow acts as the orchestrator to coordinate the transactions. If the workflow contains any errors, the orchestrator runs the compensatory transactions to ensure that data integrity is maintained across services.

The following diagram shows the steps that run inside the Step Functions workflow. The `Place Order`, `Update Inventory`, and `Make Payment` steps indicate the success path. The order is placed, the inventory is updated, and the payment is processed before a `Success` state is returned to the caller.

The `Revert Payment`, `Revert Inventory`, and `Remove Order` Lambda functions indicate the compensatory transactions that the orchestrator runs when any step in the workflow fails. If the workflow fails at the `Update Inventory` step, the orchestrator calls the `Revert Inventory` and `Remove Order` steps before returning a `Fail` state to the caller. These compensatory transactions ensure that data integrity is maintained. The inventory returns to its original level and the order is reverted.



Sample code

The following sample code shows how you can create a saga orchestrator by using Step Functions. To view the complete code, see the GitHub repository for this example.

Task definitions

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

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

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

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

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

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

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

var waitState = new Wait(this, "Wait state", new WaitProps
{
    Time = WaitTime.Duration(Duration.Seconds(30))
}).Next(revertInventoryTask);
```

Step function and state machine definitions

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

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

GitHub repository

For a complete implementation of the sample architecture for this pattern, see the GitHub repository at https://github.com/aws-samples/saga-orchestration-netcore-blog.

Blog references

- Building a serverless distributed application using Saga Orchestration pattern

Related content

- Transactional outbox pattern (p. 45)

Strangler fig pattern

Intent

The strangler fig pattern helps migrate a monolithic application to a microservices architecture incrementally, with reduced transformation risk and business disruption.

Motivation

Monolithic applications are developed to provide most of their functionality within a single process or container. The code is tightly coupled. As a result, application changes require thorough retesting to avoid regression issues. The changes cannot be tested in isolation, which impacts the cycle time. As the application is enriched with more features, high complexity can lead to more time spent on maintenance, increased time to market, and, consequently, slow product innovation.

When the application scales in size, it increases the cognitive load on the team and can cause unclear team ownership boundaries. Scaling individual features based on the load isn't possible—the entire application has to be scaled to support peak load. As the systems age, the technology can become obsolete, which drives up support costs. Monolithic, legacy applications follow best practices that were available at the time of development and weren't designed to be distributed.

When a monolithic application is migrated into a microservices architecture, it can be split into smaller components. These components can scale independently, can be released independently, and can be owned by individual teams. This results in a higher velocity of change, because changes are localized and can be tested and released quickly. Changes have a smaller scope of impact because components are loosely coupled and can be deployed individually.

Replacing a monolith completely with a microservices application by rewriting or refactoring the code is a huge undertaking and a big risk. A big bang migration, where the monolith is migrated in a single operation, introduces transformation risk and business disruption. While the application is being refactored, it is extremely hard or even impossible to add new features.

One way to resolve this issue is to use the strangler fig pattern, which was introduced by Martin Fowler. This pattern involves moving to microservices by gradually extracting features and creating a new application around the existing system. The features in the monolith are replaced by microservices gradually, and application users are able to use the newly migrated features progressively. When all features are moved out to the new system, the monolithic application can be decommissioned safely.

Applicability

Use the strangler fig pattern when:

- You want to migrate your monolithic application gradually to a microservices architecture.
- A big bang migration approach is risky because of the size and complexity of the monolith.
- The business wants to add new features and cannot wait for the transformation to be complete.
- End users must be minimally impacted during the transformation.

Issues and considerations

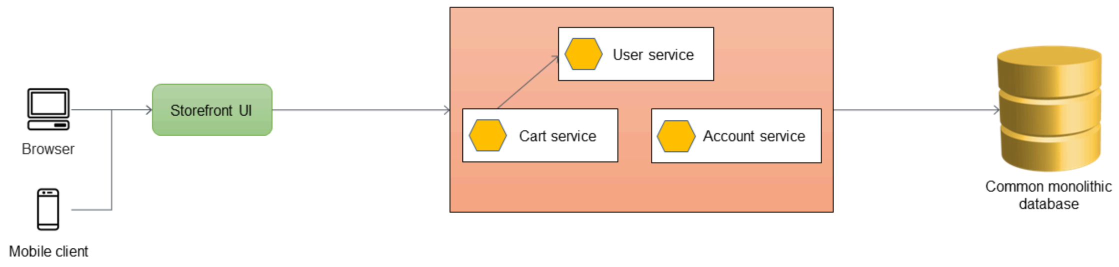
- **Code base access:** To implement the strangler fig pattern, you must have access to the monolith application's code base. As features are migrated out of the monolith, you will need to make minor code changes and implement an anti-corruption layer within the monolith to route calls to new microservices. You cannot intercept calls without code base access. Code base access is also critical for redirecting incoming requests—some code refactoring might be required so that the proxy layer can intercept the calls for migrated features and route them to microservices.
- **Unclear domain:** The premature decomposition of systems can be costly, especially when the domain isn't clear, and it's possible to get the service boundaries wrong. Domain-driven design (DDD) is a mechanism for understanding the domain, and event storming is a technique for determining domain boundaries.
- **Identifying microservices:** You can use DDD as a key tool for identifying microservices. To identify microservices, look for the natural divisions between service classes. Many services will own their own data access object and will decouple easily. Services that have related business logic and classes that have no or few dependencies are good candidates for microservices. You can refactor code before breaking down the monolith to prevent tight coupling. You should also consider compliance requirements, the release cadence, the geographical location of the teams, scaling needs, use case-driven technology needs, and the cognitive load of teams.
- **Anti-corruption layer:** During the migration process, when the features within the monolith have to call the features that were migrated as microservices, you should implement an anti-corruption layer (ACL) that routes each call to the appropriate microservice. In order to decouple and prevent changes to existing callers within the monolith, the ACL works as an adapter or a facade that converts the calls into the newer interface. This is discussed in detail in the Implementation section (p. 4) of the ACL pattern earlier in this guide.
- **Proxy layer failure:** During migration, a proxy layer intercepts the requests that go to the monolithic application and routes them to either the legacy system or the new system. However, this proxy layer can become a single point of failure or a performance bottleneck.
- **Application complexity:** Large monoliths benefit the most from the strangler fig pattern. For small applications, where the complexity of complete refactoring is low, it might be more efficient to rewrite the application in microservices architecture instead of migrating it.
- **Service interactions:** Microservices can communicate synchronously or asynchronously. When synchronous communication is required, consider whether the timeouts can cause connection or thread pool consumption, resulting in application performance issues. In such cases, use the circuit breaker pattern (p. 14) to return immediate failure for operations that are likely to fail for extended periods of time. Asynchronous communication can be achieved by using events and messaging queues.
- **Data aggregation:** In a microservices architecture, data is distributed across databases. When data aggregation is required, you can use AWS AppSync in the front end, or the command query responsibility segregation (CQRS) pattern in the backend.
- **Data consistency:** The microservices own their data store, and the monolithic application can also potentially use this data. To enable sharing, you can synchronize the new microservices' data store with the monolithic application's database by using a queue and agent. However, this can cause data redundancy and eventual consistency between two data stores, so we recommend that you treat it as a tactical solution until you can establish a long-term solution such as a data lake.

Implementation

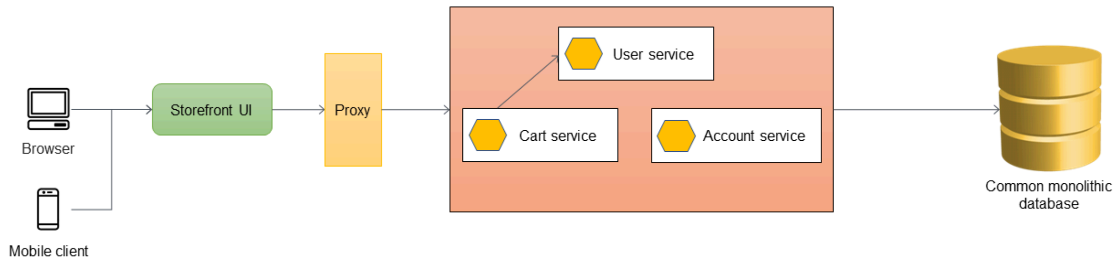
In the strangler fig pattern, you replace specific functionality with a new service or application, one component at a time. A proxy layer intercepts requests that go to the monolithic application and routes them to either the legacy system or the new system. Because the proxy layer routes users to the correct application, you can add features to the new system while ensuring that the monolith continues to function. The new system eventually replaces all the features of the old system, and you can decommission it.

High-level architecture

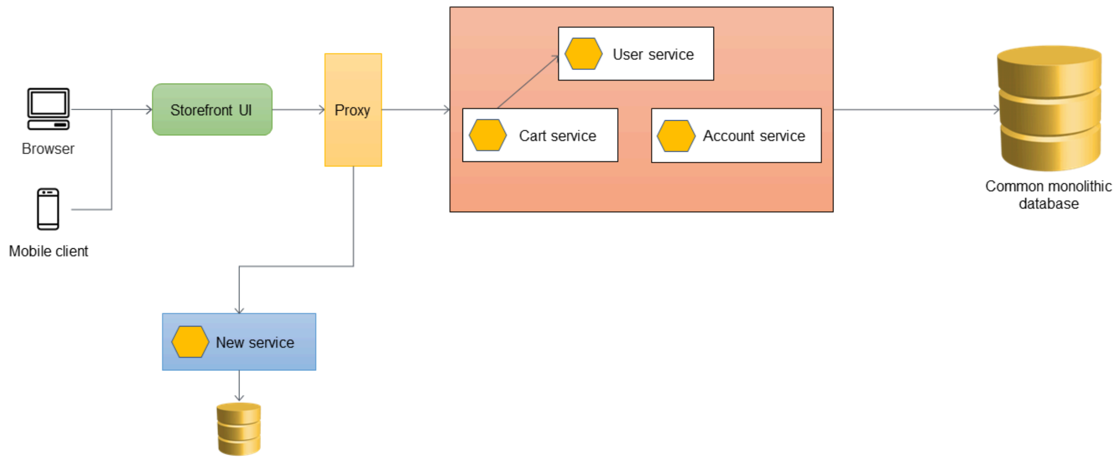
In the following diagram, a monolithic application has three services: user service, cart service, and account service. The cart service depends on the user service, and the application uses a monolithic relational database.



The first step is to add a proxy layer between the storefront UI and the monolithic application. At the start, the proxy routes all traffic to the monolithic application.

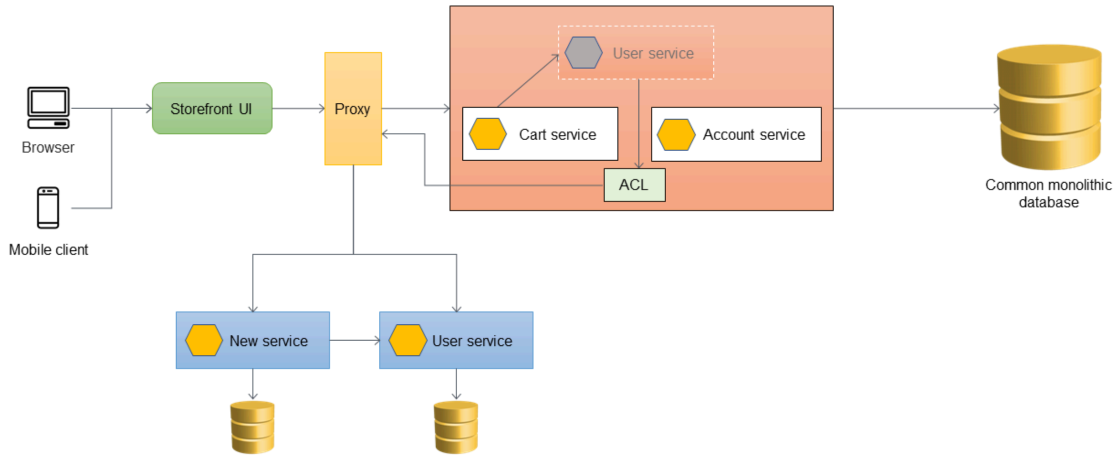


When you want to add new features to your application, you implement them as new microservices instead of adding features to the existing monolith. However, you continue to fix bugs in the monolith to ensure application stability. In the following diagram, the proxy layer routes the calls to the monolith or to the new microservice based on the API URL.



Adding an anti-corruption layer

In the following architecture, the user service has been migrated to a microservice. The cart service calls the user service, but the implementation is no longer available within the monolith. Also, the interface of the newly migrated service might not match its previous interface inside the monolithic application. To address these changes, you implement an ACL. During the migration process, when the features within the monolith need to call the features that were migrated as microservices, the ACL converts the calls to the new interface and routes them to the appropriate microservice.

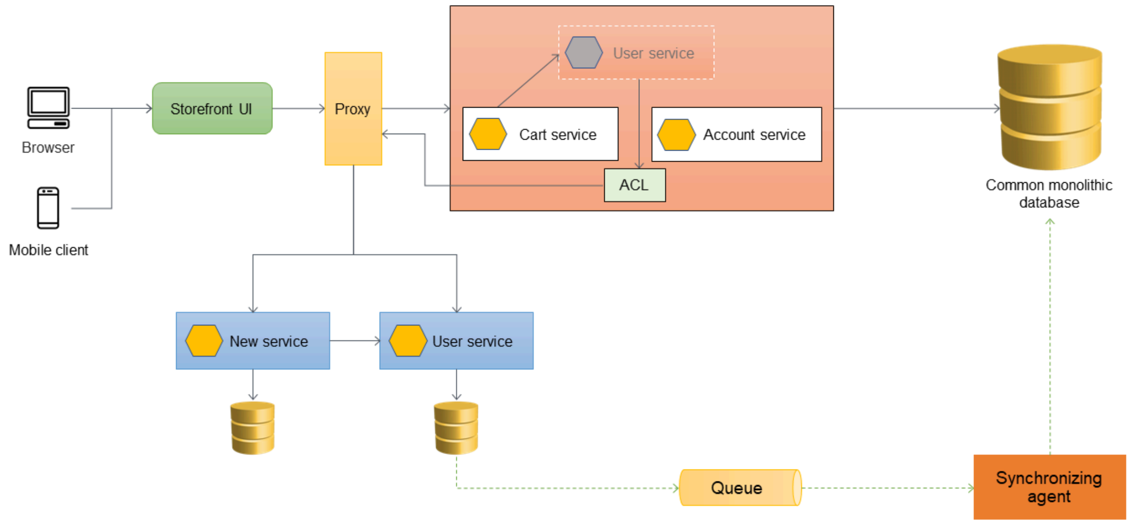


You can implement the ACL inside the monolithic application as a class that's specific to the service that was migrated; for example, `UserServiceFacade` or `UserServiceAdapter`. The ACL must be decommissioned after all dependent services have been migrated into the microservices architecture.

When you use the ACL, the cart service still calls the user service within the monolith, and the user service redirects the call to the microservice through the ACL. The cart service should still call the user service without being aware of the microservice migration. This loose coupling is required to reduce regression and business disruption.

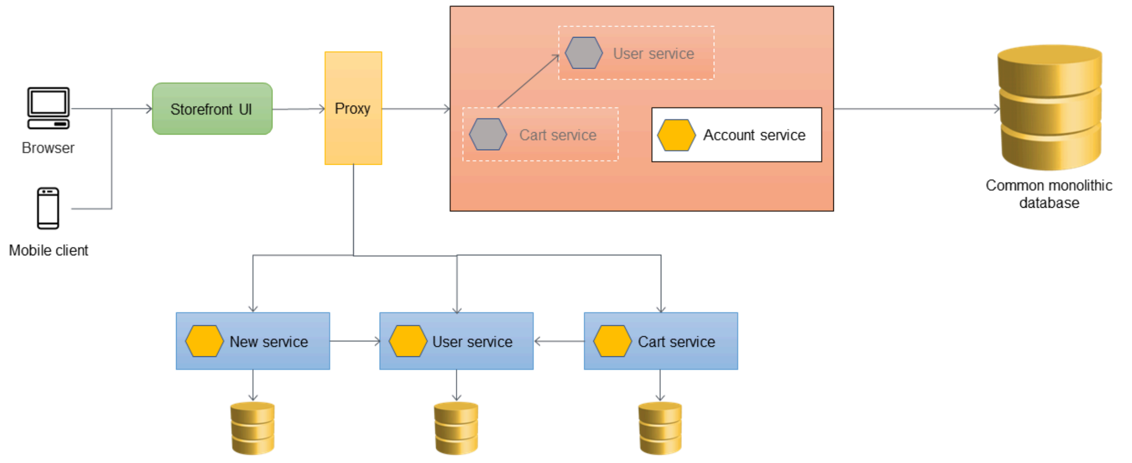
Handling data synchronization

As a best practice, the microservice should own its data. The user service stores its data in its own data store. It might need to synchronize data with the monolithic database to handle dependencies such as reporting and to support downstream applications that are not yet ready to access the microservices directly. The monolithic application might also require the data for other functions and components that haven't been migrated to microservices yet. So data synchronization is necessary between the new microservice and the monolith. To synchronize the data, you can introduce a synchronizing agent between the user microservice and the monolithic database, as shown in the following diagram. The user microservice sends an event to the queue whenever its database is updated. The synchronizing agent listens to the queue and continuously updates the monolithic database. The data in the monolithic database is eventually consistent for the data that is being synchronized.

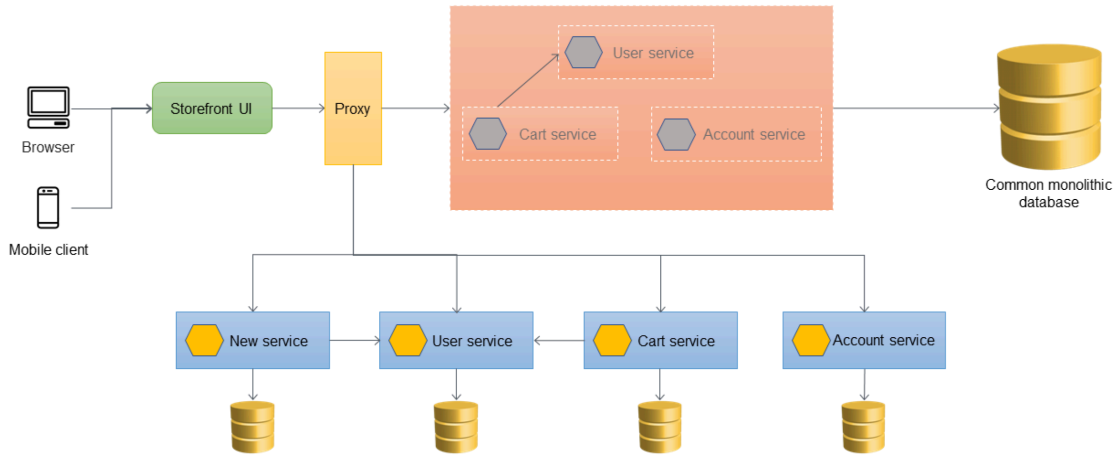


Migrating additional services

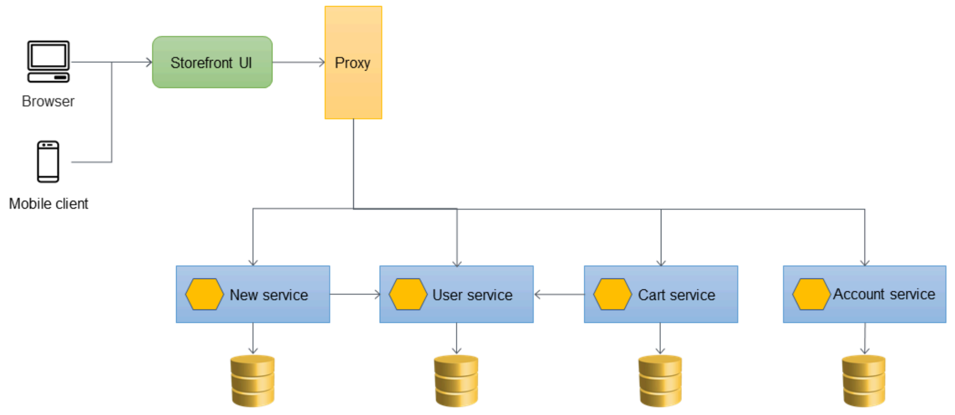
When the cart service is migrated out of the monolithic application, its code is revised to call the new service directly, so the ACL no longer routes those calls. The following diagram illustrates this architecture.



The following diagram shows the final strangled state where all services have been migrated out of the monolith and only the skeleton of the monolith remains. Historical data can be migrated to data stores owned by individual services. The ACL can be removed, and the monolith is ready to be decommissioned at this stage.



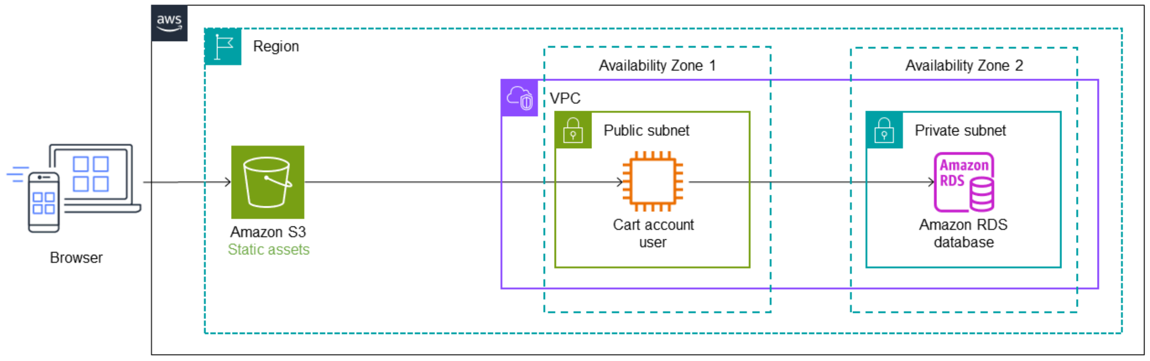
The following diagram shows the final architecture after the monolithic application has been decommissioned. You can host the individual microservices through a resource-based URL (such as `http://www.storefront.com/user`) or through their own domain (for example, `http://user.storefront.com`) based on your application's requirements. For more information about the major methods for exposing HTTP APIs to upstream consumers by using hostnames and paths, see the API routing patterns (p. 8) section.



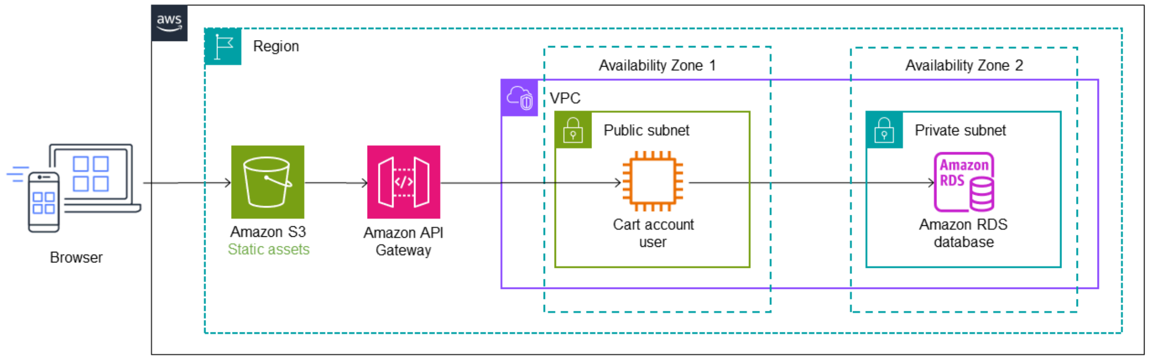
Implementation using AWS services

Using API Gateway as the application proxy

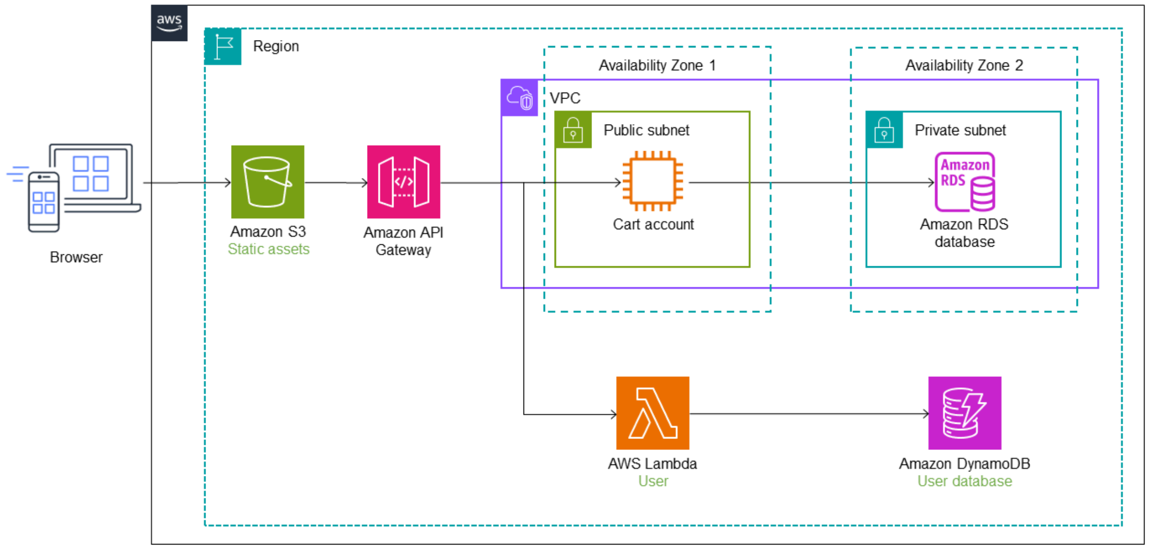
The following diagram shows the initial state of the monolithic application. Let's assume that it was migrated to AWS by using a lift-and-shift strategy, so it's running on an Amazon Elastic Compute Cloud (Amazon EC2) instance and uses an Amazon Relational Database Service (Amazon RDS) database. For simplicity, the architecture uses a single virtual private cloud (VPC) with one private and one public subnet, and let's assume that the microservices will initially be deployed within the same AWS account. (The best practice in production environments is to use a multi-account architecture to ensure deployment independence.) The EC2 instance resides in a single Availability Zone in the public subnet, and the RDS instance resides in a single Availability Zone in the private subnet. Amazon Simple Storage Service (Amazon S3) stores static assets such as the JavaScript, CSS, and React files for the website.



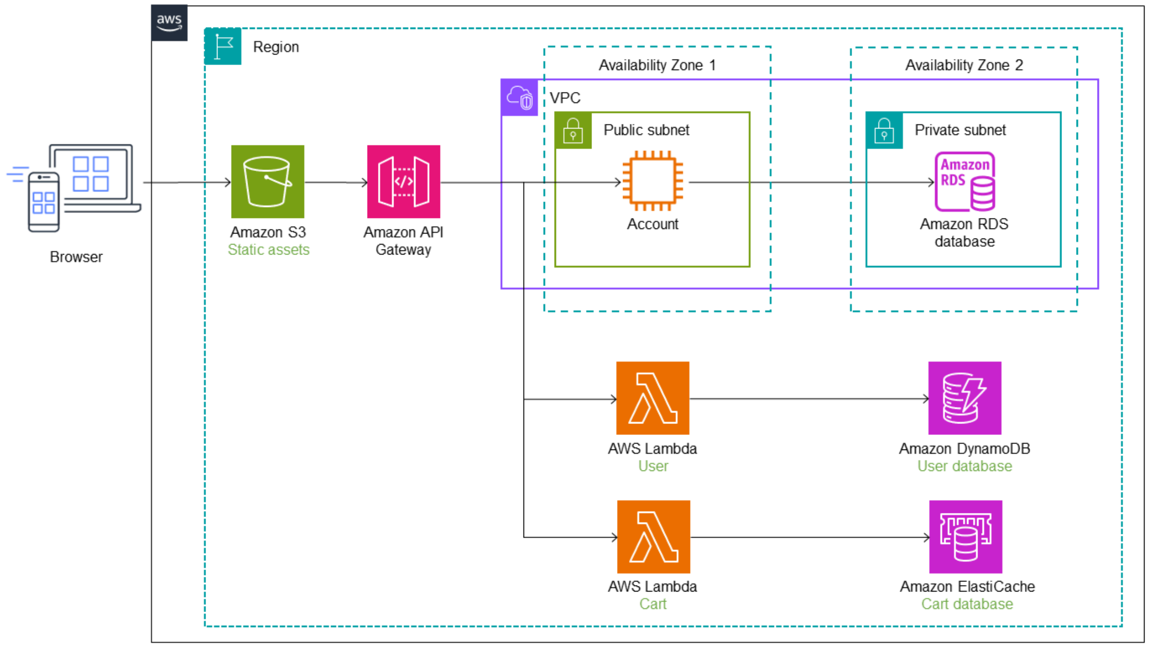
In the following architecture, AWS Migration Hub Refactor Spaces deploys Amazon API Gateway in front of the monolithic application. Refactor Spaces creates a refactoring infrastructure inside your account, and API Gateway acts as the proxy layer for routing calls to the monolith. Initially, all calls are routed to the monolithic application through the proxy layer. As discussed earlier, proxy layers can become a single point of failure. However, using API Gateway as the proxy mitigates the risk because it is a serverless, Multi-AZ service.



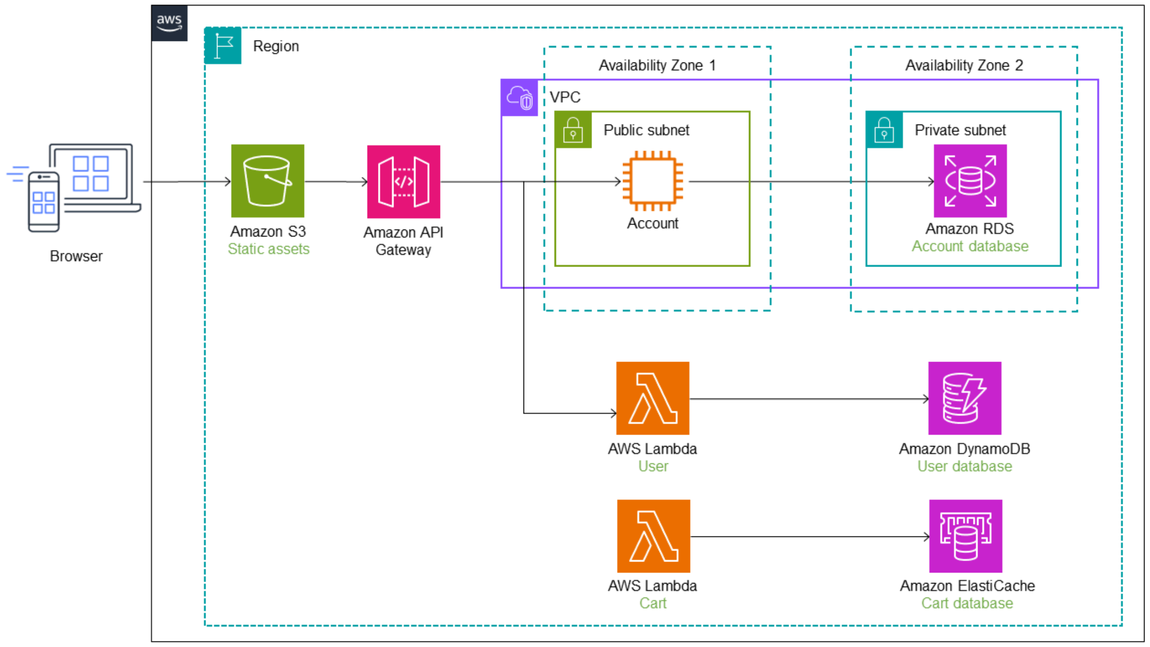
The user service is migrated into a Lambda function, and an Amazon DynamoDB database stores its data. A Lambda service endpoint and default route are added to Refactor Spaces, and API Gateway is automatically configured to route the calls to the Lambda function. For implementation details, see Module 2 in the Iterative App Modernization Workshop.



In the following diagram, the cart service has also been migrated out of the monolith and into a Lambda function. An additional route and service endpoint are added to Refactor Spaces, and traffic automatically cuts over to the `Cart` Lambda function. The data store for the Lambda function is managed by Amazon ElastiCache. The monolithic application still remains in the EC2 instance along with the Amazon RDS database.



In the next diagram, the last service (account) is migrated out of the monolith into a Lambda function. It continues to use the original Amazon RDS database. The new architecture now has three microservices with separate databases. Each service uses a different type of database. This concept of using purpose-built databases to meet the specific needs of microservices is called *polyglot persistence*. The Lambda functions can also be implemented in different programming languages, as determined by the use case. During refactoring, Refactor Spaces automates the cutover and routing of traffic to Lambda. This saves your builders the time needed to architect, deploy, and configure the routing infrastructure.

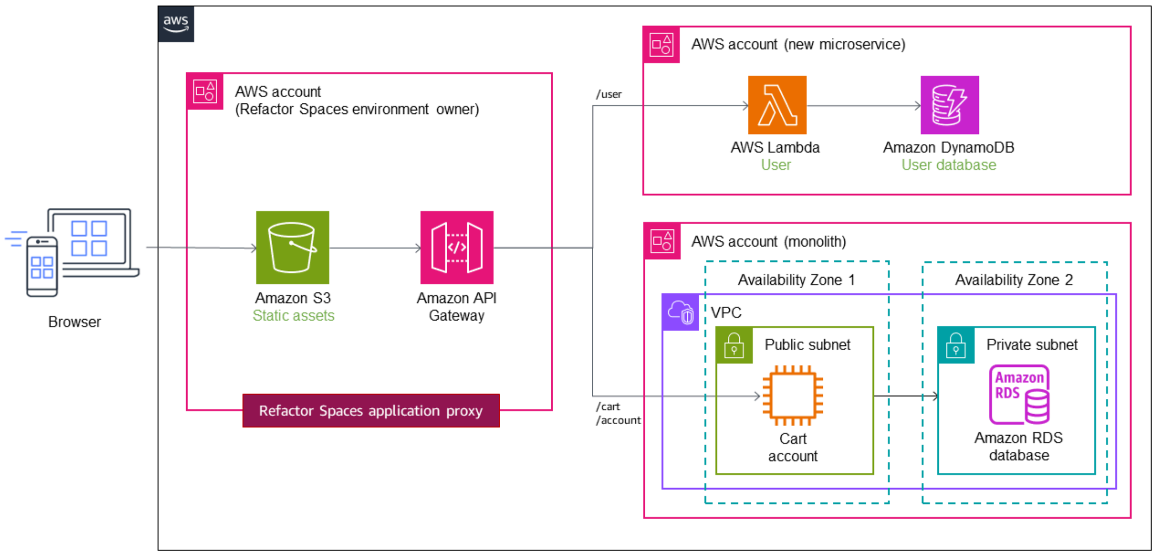


Using multiple accounts

In the previous implementation, we used a single VPC with a private and a public subnet for the monolithic application, and we deployed the microservices within the same AWS account for the sake of simplicity. However, this is rarely the case in real-world scenarios, where microservices are often deployed in multiple AWS accounts for deployment independence. In a multi-account structure, you need to configure routing traffic from the monolith to the new services in different accounts.

Refactor Spaces helps you create and configure the AWS infrastructure for routing API calls away from the monolithic application. Refactor Spaces orchestrates API Gateway, Network Load Balancer, and resource-based AWS Identity and Access Management (IAM) policies inside your AWS accounts as part of its application resource. You can transparently add new services in a single AWS account or across multiple accounts to an external HTTP endpoint. All of these resources are orchestrated inside your AWS account and can be customized and configured after deployment.

Let's assume that the user and cart services are deployed to two different accounts, as shown in the following diagram. When you use Refactor Spaces, you only need to configure the service endpoint and the route. Refactor Spaces automates the API Gateway–Lambda integration and the creation of Lambda resource policies, so you can focus on safely refactoring services off the monolith.



For a video tutorial on using Refactor Spaces, see Refactor Apps Incrementally with AWS Migration Hub Refactor Spaces.

Workshop

- Iterative app modernization workshop

Blog references

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

Related content

- API routing patterns (p. 8)
- Refactor Spaces documentation

Transactional outbox pattern

Intent

The transactional outbox pattern resolves the dual write operations issue that occurs in distributed systems when a single operation involves both a database write operation and a message or event notification. A dual write operation occurs when an application writes to two different systems; for example, when a microservice needs to persist data in the database and send a message to notify other systems. A failure in one of these operations might result in inconsistent data.

Motivation

When a microservice sends an event notification after a database update, these two operations should run atomically to ensure data consistency and reliability.

- If the database update is successful but the event notification fails, the downstream service will not be aware of the change, and the system can enter an inconsistent state.
- If the database update fails but the event notification is sent, data could get corrupted, which might affect the reliability of the system.

Applicability

Use the transactional outbox pattern when:

- You're building an event-driven application where a database update initiates an event notification .
- You want to ensure atomicity in operations that involve two services.
- You want to implement the event sourcing pattern.

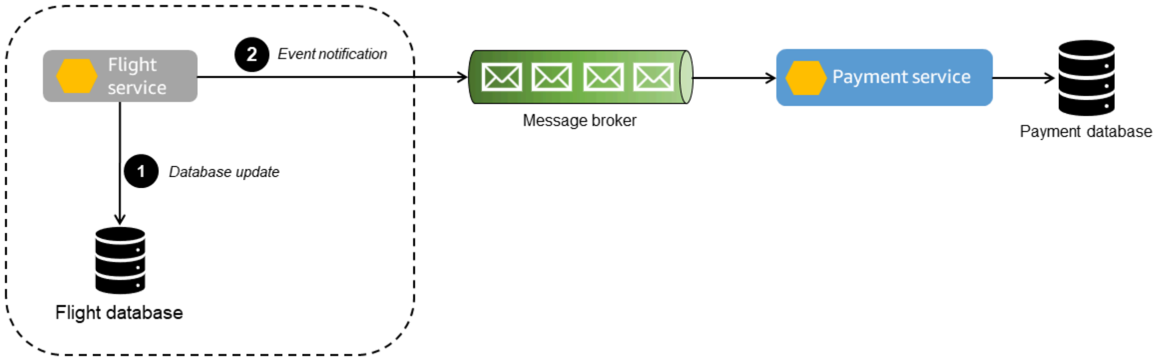
Issues and considerations

- **Duplicate messages**: The events processing service might send out duplicate messages or events, so we recommend that you make the consuming service idempotent by tracking the processed messages.
- **Order of notification**: Send messages or events in the same order in which the service updates the database. This is critical for the event sourcing pattern where you can use an event store for point-in-time recovery of the data store. If the order is incorrect, it might compromise the quality of the data. Eventual consistency and database rollback can compound the issue if the order of notifications isn't preserved.
- **Transaction rollback**: Do not send out an event notification if the transaction is rolled back.
- **Service-level transaction handling**: If the transaction spans services that require data store updates, use the saga orchestration pattern (p. 27) to preserve data integrity across the data stores.

Implementation

High-level architecture

The following sequence diagram shows the order of events that happen during dual write operations.



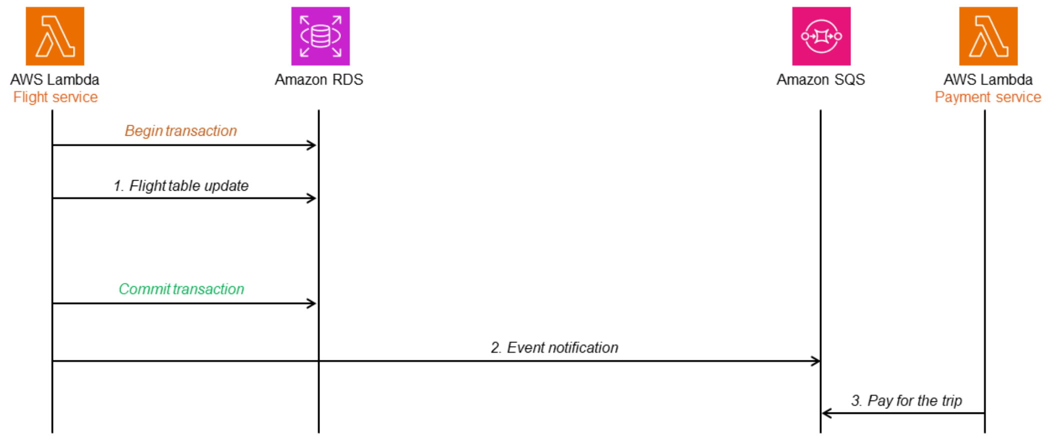
1. The flight service writes to the database and sends out an event notification to the payment service.
2. The message broker carries the messages and events to the payment service. Any failure in the message broker prevents the payment service from receiving the updates.

If the flight database update fails but the notification is sent out, the payment service will process the payment based on the event notification. This will cause downstream data inconsistencies.

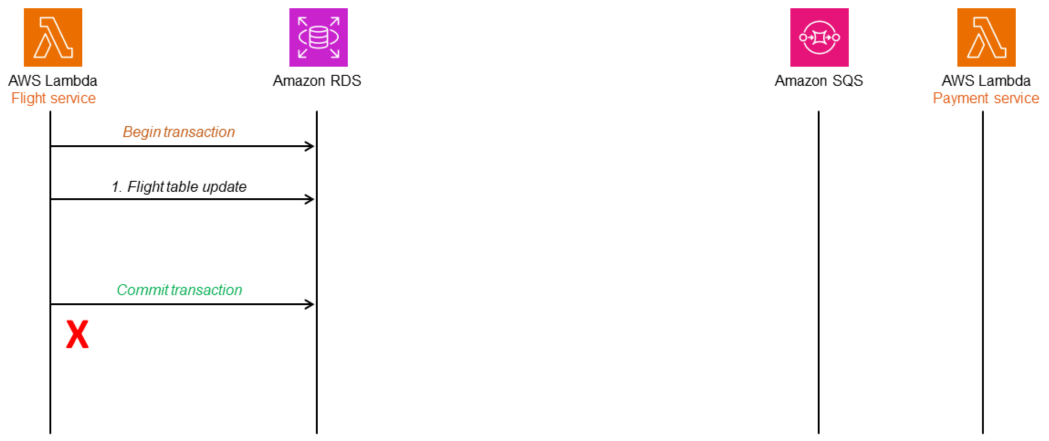
Implementation using AWS services

To demonstrate the pattern in the sequence diagram, we will use the following AWS services, as shown in the following diagram.

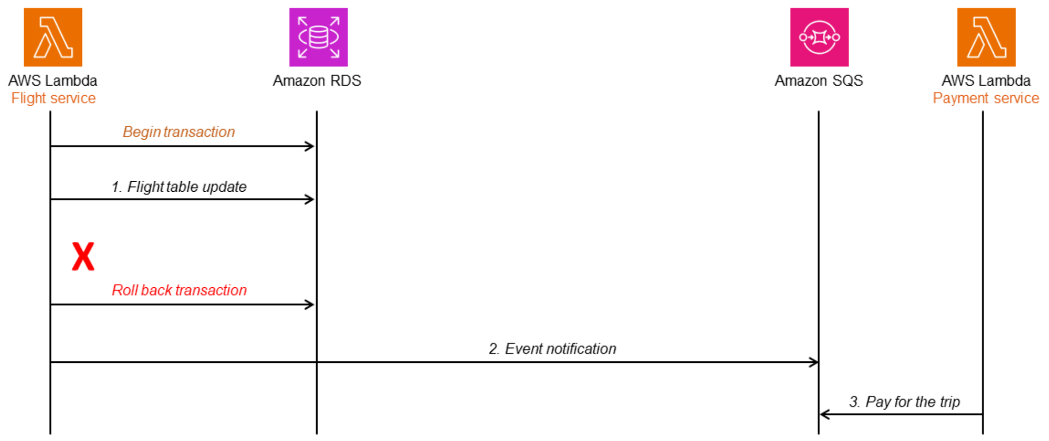
- Microservices are implemented by using AWS Lambda.
- The primary database is managed by Amazon Relational Database Service (Amazon RDS) .
- Amazon Simple Queue Service (Amazon SQS) acts as the message broker that receives event notifications.



If the flight service fails after committing the transaction, this might result in the event notification not being sent.



However, the transaction could fail and roll back, but the event notification might still be sent, causing the payment service to process the payment.



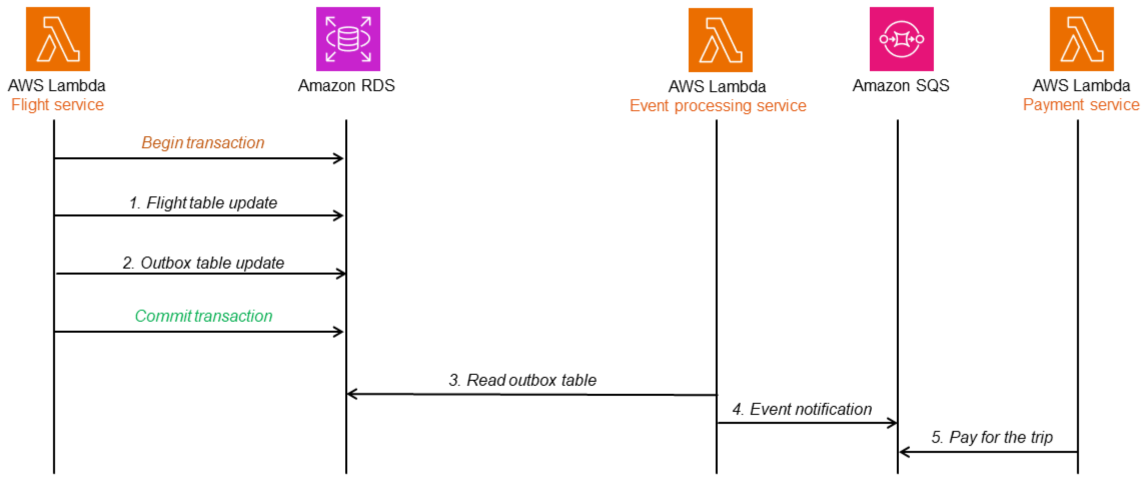
To address this problem, you can use an outbox table or change data capture (CDC). The following sections discuss these two options and how you can implement them by using AWS services.

Using an outbox table with a relational database

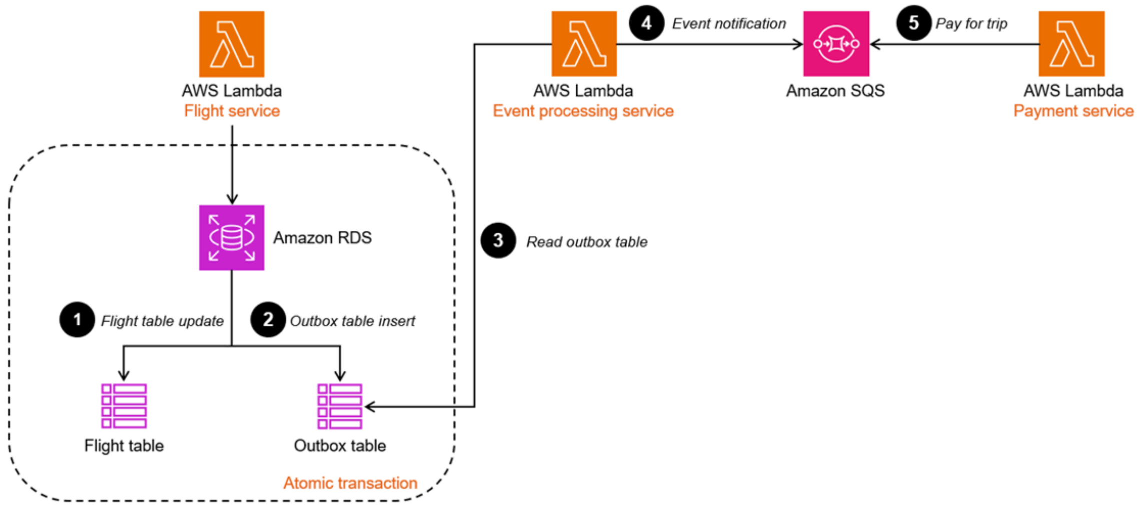
An outbox table stores all the events from the flight service with a timestamp and a sequence number.

When the flight table is updated, the outbox table is also updated in the same transaction. Another service (for example, the event processing service) reads from the outbox table and sends the event to Amazon SQS. Amazon SQS sends a message about the event to the payment service for further processing. Amazon SQS standard queues guarantee that the message is delivered at least once and doesn't get lost. However, when you use Amazon SQS standard queues, the same message or event might be delivered more than once, so you should ensure that the event notification service is idempotent (that is, processing the same message multiple times shouldn't have an adverse effect). If you require the message to be delivered exactly once, with message ordering, you can use Amazon SQS first in, first out (FIFO) queues.

If the flight table update fails or the outbox table update fails, the entire transaction is rolled back, so there are no downstream data inconsistencies.



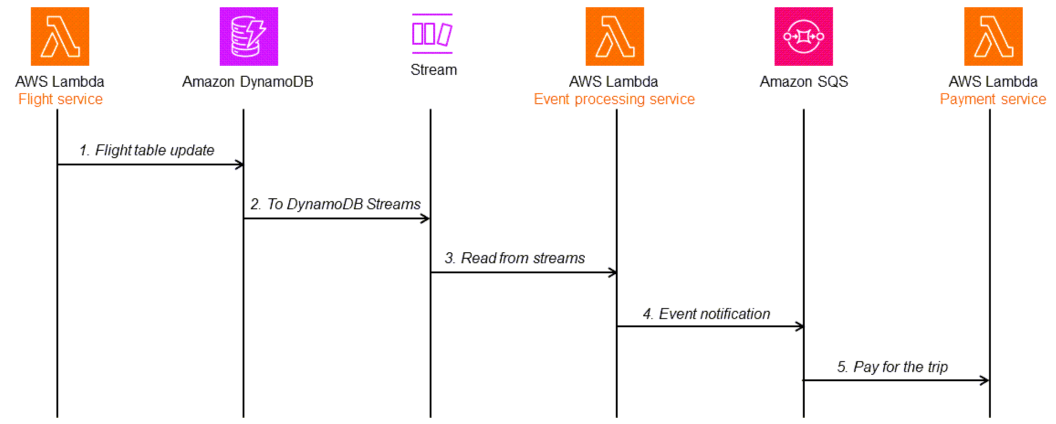
In the following diagram, the transactional outbox architecture is implemented by using an Amazon RDS database. When the events processing service reads the outbox table, it recognizes only those rows that are part of a committed (successful) transaction, and then places the message for the event in the SQS queue, which is read by the payment service for further processing. This design resolves the dual write operations issue and preserves the order of messages and events by using timestamps and sequence numbers.



Using CDC

Some databases support the publishing of item-level modifications to capture changed data. You can identify the changed items and send an event notification accordingly. This saves the overhead of creating another table to track the updates. The event initiated by the flight service is stored in another attribute of the same item.

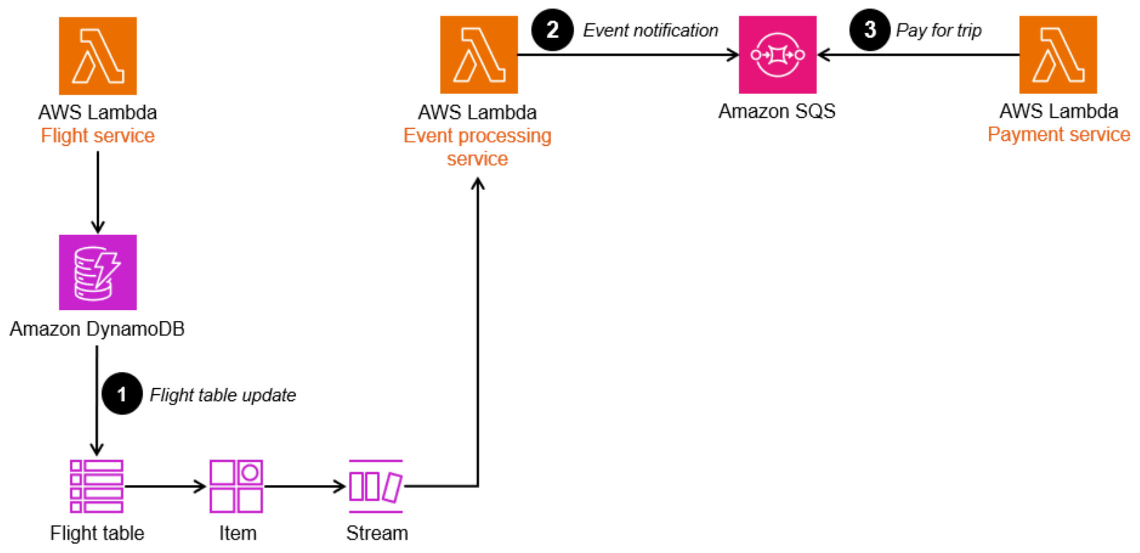
Amazon DynamoDB is a key-value NoSQL database that supports CDC updates. In the following sequence diagram, DynamoDB publishes item-level modifications to Amazon DynamoDB Streams. The event processing service reads from the streams and publishes the event notification to the payment service for further processing.



DynamoDB Streams captures the flow of information relating to item-level changes in a DynamoDB table by using a time-ordered sequence.

You can implement a transactional outbox pattern by enabling streams on the DynamoDB table. The Lambda function for the event processing service is associated with these streams.

- When the flight table is updated, the changed data is captured by DynamoDB Streams, and the events processing service polls the stream for new records.
- When new stream records become available, the Lambda function synchronously places the message for the event in the SQS queue for further processing. You can add an attribute to the DynamoDB item to capture timestamp and sequence number as needed to improve the robustness of the implementation.



Resources

References

- AWS Architecture Center
- AWS Developer Center
- The Amazon Builders Library

Tools

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

Methodologies

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

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Update (p. 51)	Updated the strangler fig pattern implementation section.	October 2, 2023
Initial publication (p. 51)	This first release includes eight design patterns: anti-corruption layer (ACL), API routing, circuit breaker, orchestration and choreography, retry with backoff, saga orchestration, strangler fig, and transactional outbox.	July 28, 2023